

Microsoft

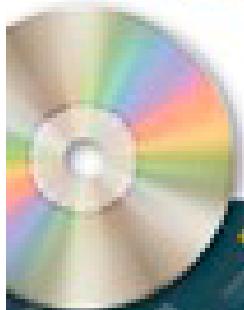
Microsoft

Visual C# 2010

en Español

Nacho Cabanes

content+master



eBook + exercises

PASO a PASO

Contenido

0. Conceptos básicos sobre programación	6
0.1. Lenguajes de alto nivel y de bajo nivel.	6
0.2. Ensambladores, compiladores e intérpretes	8
0.3. Pseudocódigo	9
1. Toma de contacto con C#	10
1.1 Escribir un texto en C#	10
1.2. Cómo probar este programa con Mono	12
1.3. Mostrar números enteros en pantalla	18
1.4. Operaciones aritméticas básicas	19
1.4.1. Orden de prioridad de los operadores	19
1.4.2. Introducción a los problemas de desbordamiento	19
1.5. Introducción a las variables: int	20
1.5.1. Definición de variables: números enteros	20
1.5.2. Asignación de valores	20
1.5.3. Mostrar el valor de una variable en pantalla	21
1.6. Identificadores	22
1.7. Comentarios	23
1.8. Datos por el usuario: ReadLine	24
2. Estructuras de control	26
2.1. Estructuras alternativas	26
2.1.1. if	26
2.1.2. if y sentencias compuestas	27
2.1.3. Operadores relacionales: <, <=, >, >=, ==, !=	28
2.1.4. if-else	29
2.1.5. Operadores lógicos: &&, , !	31
2.1.6. El peligro de la asignación en un "if"	31
2.1.7. Introducción a los diagramas de flujo	32
2.1.8. Operador condicional: ?	34
2.1.10. switch	35
2.2. Estructuras repetitivas	39
2.2.1. while	39
2.2.2. do ... while	40
2.2.3. for	42
2.3. Sentencia break: termina el bucle	46
2.4. Sentencia continue: fuerza la siguiente iteración	47
2.5. Sentencia goto	49
2.6. Más sobre diagramas de flujo. Diagramas de Chapin.	50
2.7. El caso de "foreach"	52
2.8. Recomendación de uso para los distintos tipos de bucle	52
3. Tipos de datos básicos	54
3.1. Tipo de datos entero y carácter	54
3.1.1. Tipos de datos para números enteros	54
3.1.2. Conversiones de cadena a entero	54
3.1.3. Incremento y decremento	55
3.1.4. Operaciones abreviadas: +=	56
3.2. Tipo de datos real	56
3.2.1. Simple y doble precisión	57
3.2.2. Mostrar en pantalla números reales	57
3.2.3. Formatear números	58

3.3. Tipo de datos carácter	60
3.3.1. Secuencias de escape: \n y otras.	61
3.4. Toma de contacto con las cadenas de texto	63
3.5. Los valores "booleanos"	64
4. Arrays, estructuras y cadenas de texto	66
4.1. Conceptos básicos sobre arrays o tablas	66
4.1.1. Definición de un array y acceso a los datos	66
4.1.2. Valor inicial de un array	67
4.1.3. Recorriendo los elementos de una tabla	68
4.1.4. Datos repetitivos introducidos por el usuario	69
4.2. Tablas bidimensionales	70
4.3. Estructuras o registros	73
4.3.1. Definición y acceso a los datos	73
4.3.2. Arrays de estructuras	74
4.3.3. Estructuras anidadas	75
4.4. Cadenas de caracteres	76
4.4.1. Definición. Lectura desde teclado	76
4.4.2. Cómo acceder a las letras que forman una cadena	77
4.4.3. Longitud de la cadena	77
4.4.4. Extraer una subcadena	78
4.4.5. Buscar en una cadena	78
4.4.6. Otras manipulaciones de cadenas	78
4.4.7. Comparación de cadenas	81
4.4.8. Una cadena modificable: StringBuilder	82
4.4.9. Recorriendo con "foreach"	83
4.5 Ejemplo completo	84
4.6 Ordenaciones simples	88
5. Introducción a las funciones	93
5.1. Diseño modular de programas: Descomposición modular	93
5.2. Conceptos básicos sobre funciones	93
5.3. Parámetros de una función	95
5.4. Valor devuelto por una función. El valor "void".	96
5.5. Variables locales y variables globales	98
5.6. Los conflictos de nombres en las variables	100
5.7. Modificando parámetros	101
5.8. El orden no importa	103
5.9. Algunas funciones útiles	104
5.9.1. Números aleatorios	104
5.9.2. Funciones matemáticas	105
5.9.3. Pero hay muchas más funciones...	106
5.10. Recursividad	106
5.11. Parámetros y valor de retorno de "Main"	108
6. Programación orientada a objetos	111
6.1. ¿Por qué los objetos?	111
6.2. Objetos y clases en C#	112
6.3. La herencia. Visibilidad	117
6.4. ¿Cómo se diseñan las clases?	121
6.5. La palabra "static"	122
6.6. Constructores y destructores.	123
6.7. Polimorfismo y sobrecarga	126

6.8. Orden de llamada de los constructores	126
6.9. Arrays de objetos	128
6.10. Funciones virtuales. La palabra "override"	132
6.11. Llamando a un método de la clase "padre"	136
6.12. La palabra "this": el objeto actual	138
6.13. Sobre carga de operadores	139
6.14. Proyectos a partir de varios fuentes	139
7. Manejo de ficheros	145
7.1. Escritura en un fichero de texto	145
7.2. Lectura de un fichero de texto	146
7.3. Lectura hasta el final del fichero	147
7.4. Añadir a un fichero existente	148
7.5. Ficheros en otras carpetas	149
7.6. Saber si un fichero existe	149
7.7. Más comprobaciones de errores: excepciones	150
7.8. Conceptos básicos sobre ficheros	153
7.9. Leer datos básicos de un fichero binario	154
7.10. Leer bloques de datos de un fichero binario	155
7.11. La posición en el fichero	156
7.12. Escribir en un fichero binario	158
7.13. Ejemplo: leer información de un fichero BMP	162
7.14. Leer y escribir en un mismo fichero binario	166
8. Punteros y gestión dinámica de memoria	169
8.1. ¿Por qué usar estructuras dinámicas?	169
8.2. Una pila en C#	170
8.3. Una cola en C#	171
8.4. Las listas	172
8.4.1. ArrayList	172
8.4.2. SortedList	175
8.5. Las "tablas hash"	176
8.6. Los "enumeradores"	178
8.7. Cómo "imitar" una pila usando "arrays"	180
8.8. Los punteros en C#.	182
8.8.1. ¿Qué es un puntero?	182
8.8.2. Zonas "inseguras": unsafe	182
8.8.3. Uso básico de punteros	183
8.8.4. Zonas inseguras	184
8.8.4. Reservar espacio: stackalloc	185
8.8.5. Aritmética de punteros	185
8.8.6. La palabra "fixed"	187
9. Otras características avanzadas de C#	189
9.1. Espacios de nombres	189
9.2. Operaciones con bits	191
9.3. Enumeraciones	192
9.4. Propiedades	194
9.5. Parámetros de salida (out)	196

9.6. Introducción a las expresiones regulares.	196
9.7. El operador coma	199
9.8. Lo que no vamos a ver...	200
10. Algunas bibliotecas adicionales de uso frecuente	201
10.1. Más posibilidades de la "consola"	201
10.2. Nociones básicas de entornos gráficos	203
10.3. Usando ventanas predefinidas	207
10.4. Una aplicación con dos ventanas	209
10.5. Dibujando con Windows Forms	212
10.6. Fecha y hora. Temporización	213
10.7. Lectura de directorios	215
10.8. El entorno. Llamadas al sistema	216
10.9. Datos sobre "el entorno"	217
10.10. Acceso a bases de datos con SQLite	217
10.11. Juegos con Tao.SDL	219
10.11.1. Mostrar una imagen estática	220
10.11.2. Una imagen que se mueve con el teclado	222
10.11.3. Escribir texto	224
10.11.4. Imágenes PNG y JPG	225
10.11.5. Un fuente más modular: el "bucle de juego"	226
10.11.6. Varias clases auxiliares	229
10.12. Algunos servicios de red	232
11. Depuración, prueba y documentación de programas	236
11.1. Conceptos básicos sobre depuración	236
11.2. Depurando desde VS2008 Express	236
11.3. Prueba de programas	239
11.4. Documentación básica de programas	241
11.4.1. Consejos para comentar el código	242
11.4.2. Generación de documentación a partir del código fuente.	244
Apéndice 1. Unidades de medida y sistemas de numeración	247
Ap1.1. bytes, kilobytes, megabytes...	247
Ap1.2. Unidades de medida empleadas en informática (2): los bits	248
Apéndice 2. El código ASCII	250
Apéndice 3. Sistemas de numeración.	251
Ap3.1. Sistema binario	251
Ap3.2. Sistema octal	252
Ap3.3. Sistema hexadecimal	254
Ap3.4. Representación interna de los enteros negativos	255
Apéndice 4. Instalación de Visual Studio.	258
Ap4.1. Visual Studio 2008 Express	258
Ap4.2. Visual Studio 2010 Express	264
Índice alfabético	267

0. Conceptos básicos sobre programación

Un **programa** es un conjunto de órdenes para un ordenador.

Estas órdenes se le deben dar en un cierto **lenguaje**, que el ordenador sea capaz de comprender.

El problema es que los lenguajes que realmente entienden los ordenadores resultan difíciles para nosotros, porque son muy distintos de los que nosotros empleamos habitualmente para hablar. Escribir programas en el lenguaje que utiliza internamente el ordenador (llamado "**lenguaje máquina**" o "código máquina") es un trabajo duro, tanto a la hora de crear el programa como (especialmente) en el momento de corregir algún fallo o mejorar lo que se hizo.

Por ejemplo, un programa que simplemente guardara un valor "2" en la posición de memoria 1 de un ordenador sencillo, con una arquitectura propia de los años 80, basada en el procesador Z80 de 8 bits, sería así en código máquina:

```
0011 1110 0000 0010 0011 1010 0001 0000
```

Prácticamente ilegible. Por eso, en la práctica se emplean lenguajes más parecidos al lenguaje humano, llamados "**lenguajes de alto nivel**". Normalmente, estos son muy parecidos al idioma inglés, aunque siguen unas reglas mucho más estrictas.

0.1. Lenguajes de alto nivel y de bajo nivel.

Vamos a ver en primer lugar algún ejemplo de lenguaje de alto nivel, para después comparar con lenguajes de bajo nivel, que son los más cercanos al ordenador.

Uno de los lenguajes de alto nivel más sencillos es el lenguaje BASIC. En este lenguaje, escribir el texto Hola en pantalla, sería tan sencillo como usar la orden

```
PRINT "Hola"
```

Otros lenguajes, como Pascal, nos obligan a ser algo más estrictos, pero a cambio hacen más fácil descubrir errores (ya veremos por qué):

```
program Saludo;
begin
  write('Hola');
end.
```

El equivalente en lenguaje C resulta algo más difícil de leer:

```
#include <stdio.h>

int main()
{
    printf("Hola");
}
```

En C# hay que dar todavía más pasos para conseguir lo mismo:

```
public class Ejemplo01
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

Por el contrario, los lenguajes de **bajo nivel** son más cercanos al ordenador que a los lenguajes humanos. Eso hace que sean más difíciles de aprender y también que los fallos sean más difíciles de descubrir y corregir, a cambio de que podemos optimizar al máximo la velocidad (si sabemos cómo), e incluso llegar a un nivel de control del ordenador que a veces no se puede alcanzar con otros lenguajes. Por ejemplo, escribir Hola en lenguaje ensamblador de un ordenador equipado con el sistema operativo MsDos y con un procesador de la familia Intel x86 sería algo como

```
dosseg
.model small
.stack 100h

.data
hello_message db 'Hola',0dh,0ah,'$'

.code
main proc
    mov    ax,@data
    mov    ds,ax

    mov    ah,9
    mov    dx,offset hello_message
    int    21h

    mov    ax,4C00h
    int    21h
main endp
end main
```

Resulta bastante más difícil de seguir. Pero eso todavía no es lo que el ordenador entiende, aunque tiene una equivalencia casi directa. Lo que el ordenador realmente es capaz de comprender son secuencias de ceros y unos. Por ejemplo, las órdenes "mov ds, ax" y "mov ah, 9" (en cuyo significado no vamos a entrar) se convertirían a lo siguiente:

1000 0011 1101 1000 1011 0100 0000 1001

(Nota: los colores de los ejemplos anteriores son una ayuda que nos dan algunos entornos de programación, para que nos sea más fácil descubrir errores).

0.2. Ensambladores, compiladores e intérpretes

Está claro entonces que las órdenes que nosotros hemos escrito (lo que se conoce como "programa **fuente**") deben convertirse a lo que el ordenador comprende (obteniendo el "programa **ejecutable**").

Si elegimos un lenguaje de bajo nivel, como el ensamblador (en inglés *Assembly*, abreviado como *Asm*), la traducción es sencilla, y de hacer esa traducción se encargan unas herramientas llamadas **ensambladores** (en inglés *Assembler*).

Cuando el lenguaje que hemos empleado es de alto nivel, la traducción es más complicada, y a veces implicará también recopilar varios fuentes distintos o incluir posibilidades que se encuentran en otras bibliotecas que no hemos preparado nosotros. Las herramientas encargadas de realizar todo esto son los **compiladores**.

El programa ejecutable obtenido con el compilador o el ensamblador se podría hacer funcionar en otro ordenador similar al que habíamos utilizado para crearlo, sin necesidad de que ese otro ordenador tenga instalado el compilador o el ensamblador.

Por ejemplo, en el caso de Windows (y de MsDos), y del programa que nos saluda en lenguaje Pascal, tendríamos un fichero fuente llamado SALUDO.PAS. Este fichero no serviría de nada en un ordenador que no tuviera un compilador de Pascal. En cambio, después de compilarlo obtendríamos un fichero SALUDO.EXE, capaz de funcionar en cualquier otro ordenador que tuviera el mismo sistema operativo, aunque no tenga un compilador de Pascal instalado.

Un **intérprete** es una herramienta parecida a un compilador, con la diferencia de que en los intérpretes no se crea ningún "programa ejecutable" capaz de funcionar "por sí solo", de modo que si queremos distribuir nuestro programa a alguien, deberemos entregarle el programa fuente y también el intérprete que es capaz de entenderlo, o no le servirá de nada. Cuando ponemos el programa en funcionamiento, el intérprete se encarga de convertir el programa en lenguaje de alto nivel a código máquina, orden por orden, justo en el momento en que hay que procesar cada una de las órdenes.

Hoy en día existe algo que parece intermedio entre un compilador y un intérprete. Existen lenguajes que no se compilan a un ejecutable para un ordenador concreto, sino a un ejecutable "genérico", que es capaz de funcionar en distintos tipos de ordenadores, a condición de que en ese ordenador exista una "**máquina virtual**" capaz de entender esos ejecutables genéricos. Esta es la idea que se aplica en Java: los fuentes son ficheros de texto, con extensión ".java", que se compilan a ficheros ".class". Estos ficheros ".class" se podrían llevar a cualquier ordenador que tenga instalada una "máquina virtual Java" (las hay para la mayoría de sistemas operativos). Esta misma idea se sigue en el lenguaje C#, que se apoya en una máquina virtual llamada "Dot Net Framework" (algo así como "armazón punto net").

0.3. Pseudocódigo

A pesar de que los lenguajes de alto nivel se acercan al lenguaje natural, que nosotros empleamos, es habitual no usar ningún lenguaje de programación concreto cuando queremos plantear los pasos necesarios para resolver un problema, sino emplear un lenguaje de programación ficticio, no tan estricto, muchas veces escrito incluso en español. Este lenguaje recibe el nombre de **pseudocódigo**.

Esa secuencia de pasos para resolver un problema es lo que se conoce como **algoritmo** (realmente hay alguna condición más, por ejemplo, debe ser un número finito de pasos). Por tanto, un programa de ordenador es un algoritmo expresado en un lenguaje de programación.

Por ejemplo, un algoritmo que controlase los pagos que se realizan en una tienda con tarjeta de crédito, escrito en pseudocódigo, podría ser:

```

Leer banda magnética de la tarjeta
Conectar con central de cobros
Si hay conexión y la tarjeta es correcta:
    Pedir código PIN
    Si el PIN es correcto
        Comprobar saldo_existente
        Si saldo_existente >= importe_compra
            Aceptar la venta
            Descontar importe del saldo.
        Fin Si
    Fin Si
Fin Si

```

Ejercicios propuestos

1. Localizar en Internet el intérprete de Basic llamado Bywater Basic, en su versión para el sistema operativo que se esté utilizando y probar el primer programa de ejemplo que se ha visto en el apartado 0.1.
2. Localizar en Internet el compilador de Pascal llamado Free Pascal, en su versión para el sistema operativo que se esté utilizando, instalarlo y probar el segundo programa de ejemplo que se ha visto en el apartado 0.1.
3. Localizar un compilador de C para el sistema operativo que se esté utilizando (si es Linux o alguna otra versión de Unix, es fácil que se encuentre ya instalado) y probar el tercer programa de ejemplo que se ha visto en el apartado 0.1.

1. Toma de contacto con C#

C# es un lenguaje de programación de ordenadores. Se trata de un lenguaje moderno, evolucionado a partir de C y C++, y con una sintaxis muy similar a la de Java. Los programas creados con C# no suelen ser tan rápidos como los creados con C, pero a cambio la productividad del programador es mucho mayor.

Se trata de un lenguaje creado por Microsoft para crear programas para su plataforma .NET, pero estandarizado posteriormente por ECMA y por ISO, y del que existe una implementación alternativa de "código abierto", el "proyecto Mono", que está disponible para Windows, Linux, Mac OS X y otros sistemas operativos.

Nosotros comenzaremos por usar Mono como plataforma de desarrollo durante los primeros temas. Cuando los conceptos básicos estén asentados, pasaremos a emplear Visual C#, de Microsoft, que requiere un ordenador más potente pero a cambio incluye un entorno de desarrollo muy avanzado, y está disponible también en una versión gratuita (Visual Studio Express Edition).

Los **pasos** que seguiremos para crear un programa en C# serán:

1. Escribir el programa en lenguaje C# (**fichero fuente**), con cualquier editor de textos.
2. Compilarlo con nuestro compilador. Esto creará un "**fichero ejecutable**".
3. Lanzar el fichero ejecutable.

La mayoría de los compiladores actuales permiten dar todos estos pasos desde un único **entorno**, en el que escribimos nuestros programas, los compilamos, y los depuramos en caso de que exista algún fallo.

En el siguiente apartado veremos un ejemplo de uno de estos entornos, dónde localizarlo y cómo instalarlo.

1.1 Escribir un texto en C#

Vamos con un primer ejemplo de programa en C#, posiblemente el más sencillo de los que "hacen algo útil". Se trata de escribir un texto en pantalla. La apariencia de este programa la vimos en el tema anterior. Vamos a verlo ahora con más detalle:

```
public class Ejemplo01
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

Esto escribe "Hola" en la pantalla. Pero hay mucho alrededor de ese "Hola", y vamos a comentarlo antes de proseguir, aunque muchos de los detalles se irán aclarando más adelante. En este primer análisis, iremos de dentro hacia fuera:

- `WriteLine("Hola");` - "Hola" es el texto que queremos escribir, y `WriteLine` es la orden encargada de escribir (`Write`) una línea (`Line`) de texto en pantalla.
- `Console.WriteLine("Hola");` porque `WriteLine` es una orden de manejo de la "consola" (la pantalla "negra" en modo texto del sistema operativo).
- `System.Console.WriteLine("Hola");` porque las órdenes relacionadas con el manejo de consola (`Console`) pertenecen a la categoría de sistema (`System`).
- Las llaves `{ } y }` se usan para delimitar un bloque de programa. En nuestro caso, se trata del bloque principal del programa.
- `public static void Main()` - `Main` indica cual es "el cuerpo del programa", la parte principal (un programa puede estar dividido en varios fragmentos, como veremos más adelante). Todos los programas tienen que tener un bloque "Main". Los detalles de por qué hay que poner delante "public static void" y de por qué se pone después un paréntesis vacío los iremos aclarando más tarde. De momento, deberemos memorizar que ésa será la forma correcta de escribir "Main".
- `public class Ejemplo01` - de momento pensaremos que "Ejemplo01" es el nombre de nuestro programa. Una línea como esa deberá existir también siempre en nuestros programas, y eso de "public class" será obligatorio. Nuevamente, aplazamos para más tarde los detalles sobre qué quiere decir "class" y por qué debe ser "public".

Como se puede ver, mucha parte de este programa todavía es casi un "acto de fe" para nosotros. Debemos creernos que "se debe hacer así". Poco a poco iremos detallando el por qué de "public", de "static", de "void", de "class"... Por ahora nos limitaremos a "rellenar" el cuerpo del programa para entender los conceptos básicos de programación.

Ejercicio propuesto: Crea un programa en C# que te salude por tu nombre (ej: "Hola, Nacho").

Sólo un par de cosas más antes de seguir adelante:

- Cada orden de C# debe terminar con un **punto y coma** `(;)`
- C# es un lenguaje de **formato libre**, de modo que puede haber varias órdenes en una misma línea, u órdenes separadas por varias líneas o espacios entre medias. Lo que realmente indica dónde termina una orden y donde empieza la siguiente son los puntos y coma. Por ese motivo, el programa anterior se podría haber escrito también así (aunque no es aconsejable, porque puede resultar menos legible):

```
public class Ejemplo01 {
public
static
void Main() { System.Console.WriteLine("Hola"); } }
```

- De hecho, hay dos formas especialmente frecuentes de colocar la llave de comienzo, y yo usaré ambas indistintamente. Una es como hemos hecho en el primer ejemplo: situar la llave de apertura en una línea, sola, y justo encima de la llave de cierre correspondiente. Esto es lo que muchos autores llaman el "estilo C". La segunda forma habitual es situándola a continuación del nombre del bloque que comienza (el "estilo Java"), así:

```
public class Ejemplo01 {
    public static void Main(){
        System.Console.WriteLine("Hola");
    }
}
```

(esta es la forma que yo emplearé preferentemente en este texto cuando estemos trabajando con fuentes de mayor tamaño, para que ocupe un poco menos de espacio).

- La gran mayoría de las órdenes que encontraremos en el lenguaje C# son palabras en inglés o abreviaturas de éstas. Pero hay que tener en cuenta que C# **distingue entre mayúsculas** y minúsculas, por lo que "WriteLine" es una palabra reconocida, pero "writeLine", "WRITELINE" o "Writeline" no lo son.

1.2. Cómo probar este programa con Mono

Como ya hemos comentado, usaremos Mono como plataforma de desarrollo para nuestros primeros programas. Por eso, vamos a comenzar por ver dónde encontrar esta herramienta, cómo instalarla y cómo utilizarla.

Podemos descargar Mono desde su página oficial:

<http://www.mono-project.com/>

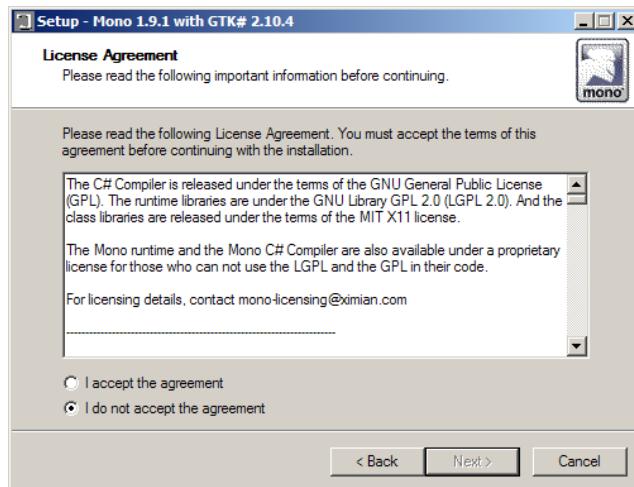
The screenshot shows the official Mono Project website. At the top, there's a navigation bar with links for 'start :: contribute :: forums'. Below the header, there's a large 'What is Mono?' section with a brief description and links to 'FAQ', 'Contacting the Mono Team', 'Bug reporting', and 'Articles and Tutorials'. To the right, there's a 'Mono in the Real World' section featuring success stories for Unity and Plastic SCM, along with a link to 'More Mono success stories...'. Further down, there's a 'Mono Project News' section with a link to 'SecondLife Launches Mono-based servers'. The right side also features links for 'screenshots', 'download now', 'manuals & docs', and 'blogs'.

En la parte superior derecha aparece el enlace para descargar ("download now"), que nos lleva a una nueva página en la que debemos elegir la plataforma para la que queremos nuestro Mono. Nosotros descargaremos la versión más reciente para Windows (la 2.6.7 en el momento de escribir este texto).

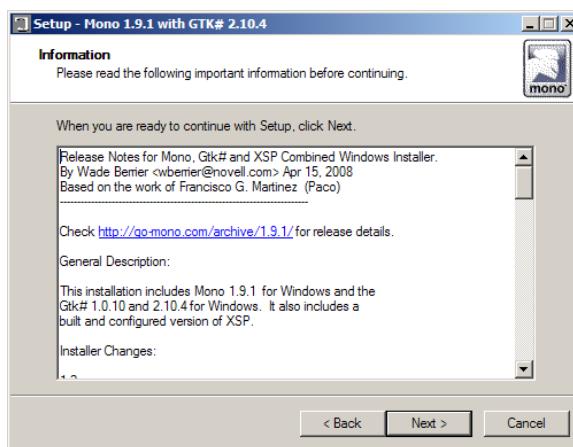
Se trata de un fichero de algo más de 70 Mb. Cuando termina la descarga, hacemos doble clic en el fichero recibido y comienza la instalación, en la que primero se nos muestra el mensaje de bienvenida:



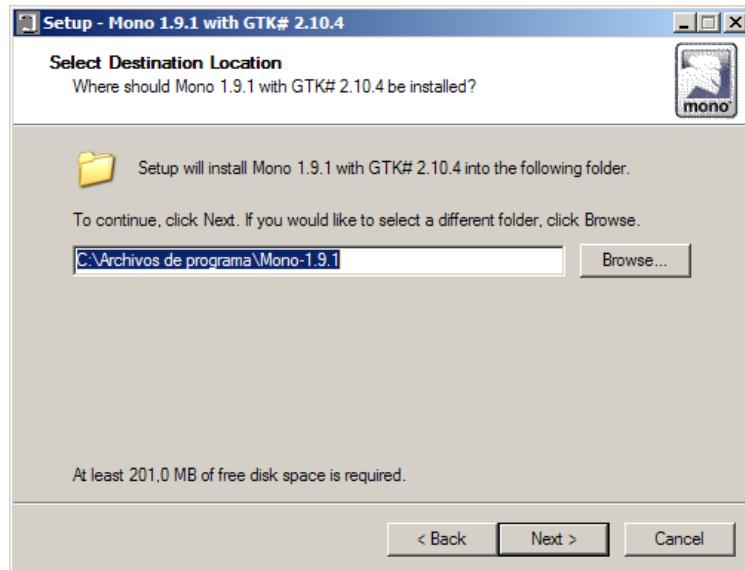
El siguiente paso será aceptar el acuerdo de licencia:



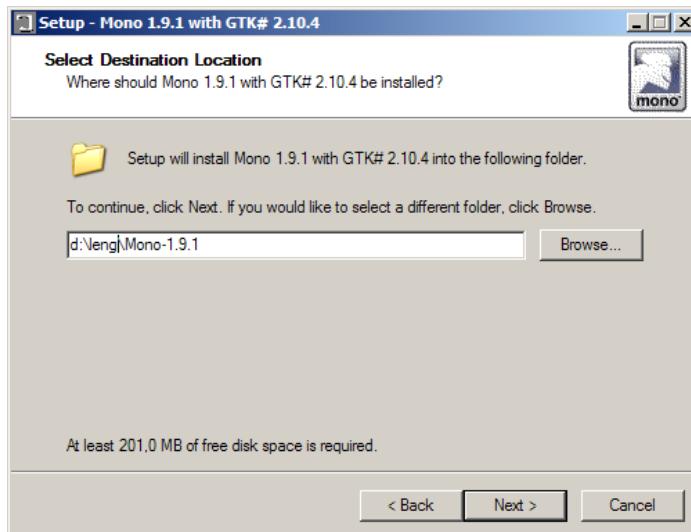
Después se nos muestra una ventana de información, en la que se nos avisa de que se va a instalar Mono 1.9.1, junto con las librerías Gtk# para creación de interfaces de usuario y XSP (eXtensible Server Pages, un servidor web).



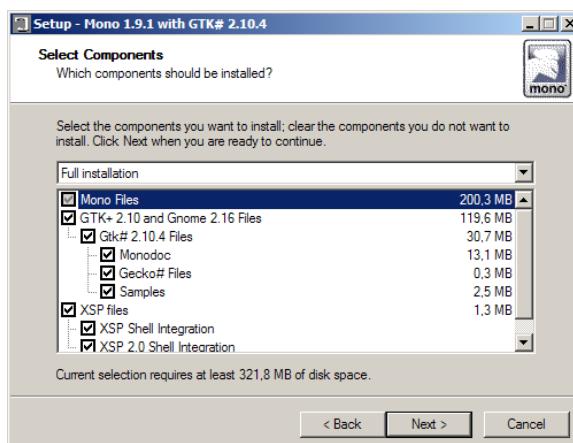
A continuación se nos pregunta en qué carpeta queremos instalar. Como es habitual, se nos propone que sea dentro de "Archivos de programa":



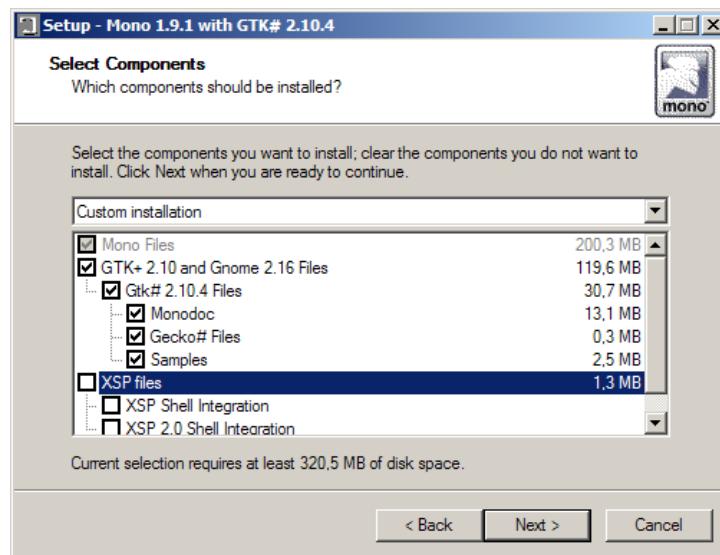
Yo no soy partidario de instalar todo en "Archivos de Programa". Mis herramientas de programación suelen estar en otra unidad de disco (D:), así que prefiero cambiar esa opción por defecto:



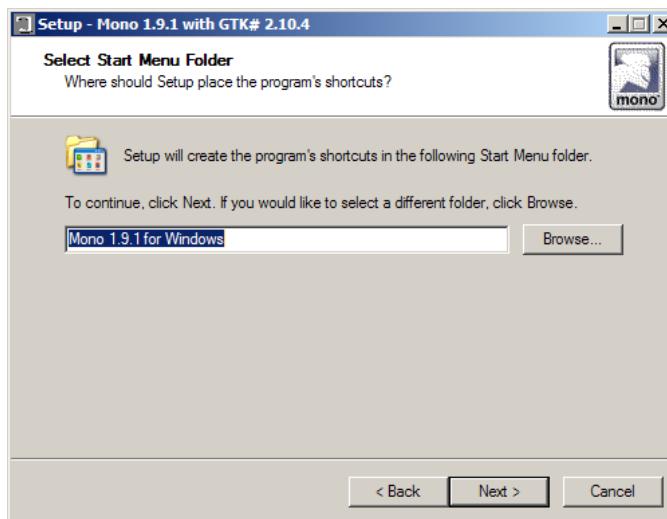
El siguiente paso es elegir qué componentes queremos instalar (Mono, Gtk#, XSP):



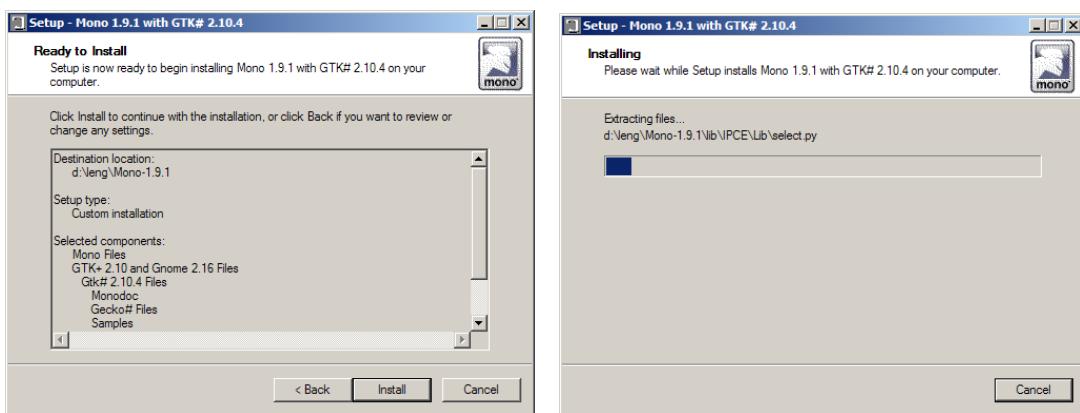
Nuevamente, soy partidario de no instalar todo. Mono es imprescindible. La creación de interfaces de usuario con Gtk# queda fuera del alcance que se pretende con este texto, pero aun así puede ser interesante para quien quiera profundizar. El servidor web XSP es algo claramente innecesario por ahora, y que además instalaría un "listener" que ralentizaría ligeramente el ordenador, así que puede ser razonable no instalarlo por ahora:



El siguiente paso es indicar en qué carpeta del menú de Inicio queremos que quede accesible:



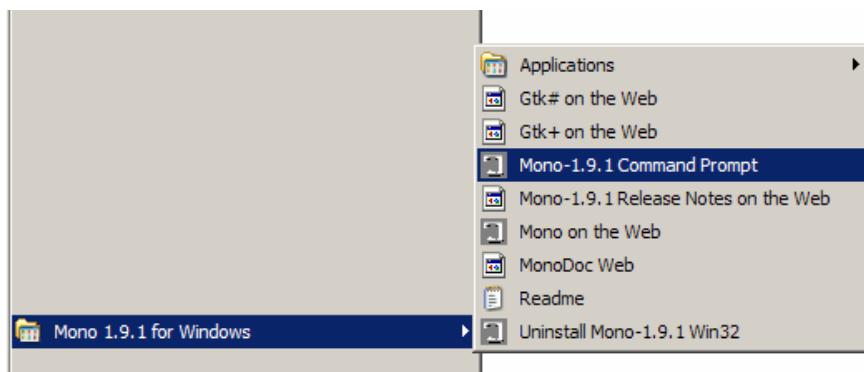
A continuación se nos muestra el resumen de lo que se va a instalar. Si confirmamos que todo nos parece correcto, comienza la copia de ficheros:



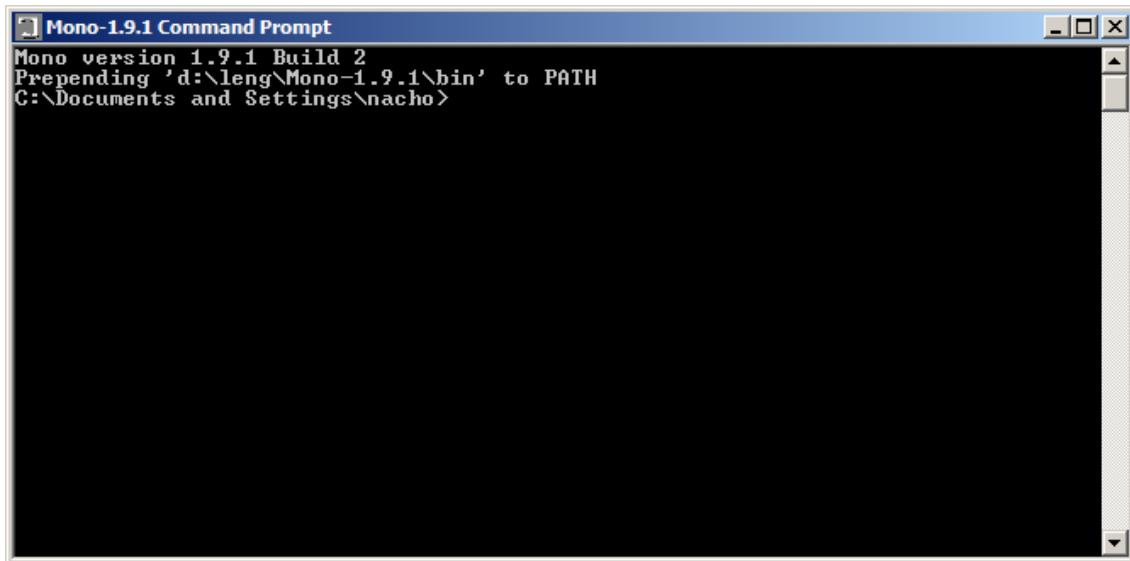
Si todo es correcto, al cabo de un instante tendremos el mensaje de confirmación de que la instalación se ha completado:



Mono está listo para usar. En nuestro menú de Inicio deberíamos tener una nueva carpeta llamada "Mono 1.9.1 for Windows", y dentro de ella un acceso a "Mono-1.9.1 Command Prompt":



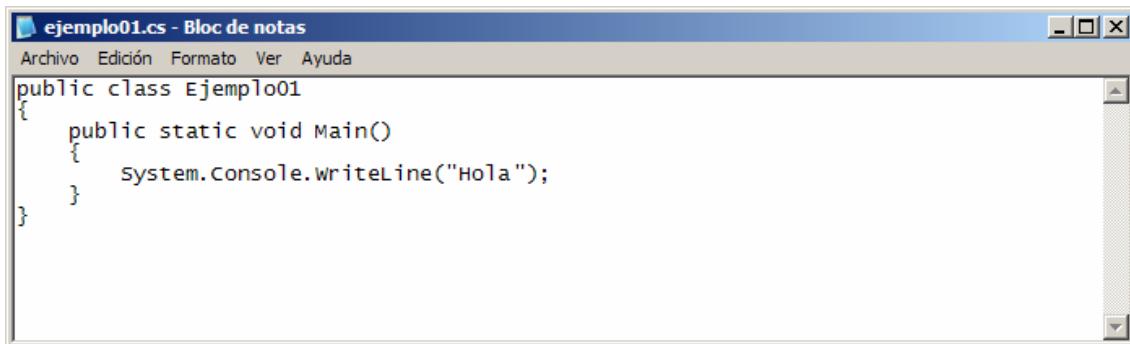
Si hacemos clic en esa opción, accedemos al símbolo de sistema ("command prompt"), la pantalla negra del sistema operativo, pero con el "path" (la ruta de búsqueda) preparada para que podamos acceder al compilador desde ella:



Primero debemos teclear nuestro fuente. Para ello podemos usar cualquier editor de texto. En este primer fuente, usaremos simplemente el "Bloc de notas" de Windows. Para ello tecleamos:

```
notepad ejemplo01.cs
```

Aparecerá la pantalla del "Bloc de notas", junto con un aviso que nos indica que no existe ese fichero, y que nos pregunta si deseamos crearlo. Respondemos que sí y podemos empezar a teclear el ejemplo que habíamos visto anteriormente:



Guardamos los cambios, salimos del "Bloc de notas" y nos volvemos a encontrar en la pantalla negra del símbolo del sistema. Nuestro fuente ya está escrito. El siguiente paso es compilarlo. Para eso, tecleamos

```
mcs ejemplo01.cs
```

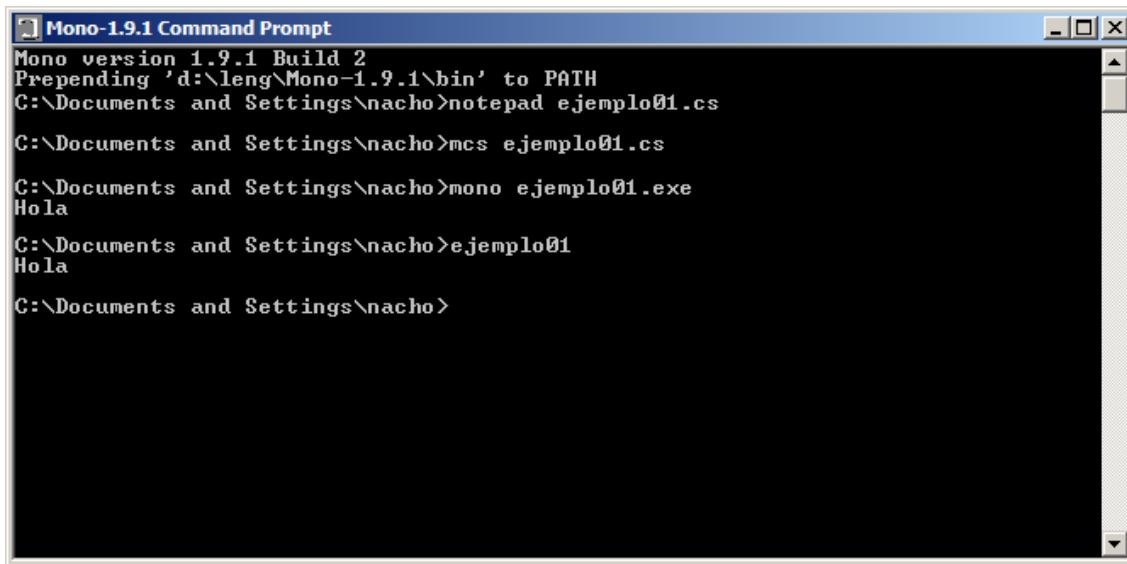
Si no se nos responde nada, quiere decir que no ha habido errores. Si todo va bien, se acaba de crear un fichero "ejemplo01.exe". En ese caso, podríamos lanzar el programa tecleando

```
mono exemplo01
```

y el mensaje "Hola" debería aparecer en pantalla.

Si en nuestro ordenador está instalado el "Dot Net Framework" (algo que debería ser cierto en las últimas versiones de Windows), no debería hacer falta decir que queremos que sea Mono quien lance nuestro programa, y podremos ejecutarlo directamente con su nombre:

```
ejemplo01
```



The screenshot shows a command prompt window titled "Mono-1.9.1 Command Prompt". The window displays the following text:

```
Mono version 1.9.1 Build 2
Prepending 'd:\leng\Mono-1.9.1\bin' to PATH
C:\Documents and Settings\nacho>notepad ejemplo01.cs
C:\Documents and Settings\nacho>mcs ejemplo01.cs
C:\Documents and Settings\nacho>mono exemplo01.exe
Hola
C:\Documents and Settings\nacho>ejemplo01
Hola
C:\Documents and Settings\nacho>
```

Nota: Si quieres un editor más potente que el Bloc de notas de Windows, puedes probar Notepad++, que es gratuito (realmente más que eso: es de "código abierto") y podrás localizar fácilmente en Internet. Y si prefieres un entorno desde el que puedas teclear, compilar y probar tus programas, incluso los de gran tamaño que estén formados por varios ficheros, en el apartado 6.13 hablaremos de SharpDevelop. Si quieres saber cosas sobre el entorno "oficial", llamado Visual Studio, los tienes en el Apéndice 4.

1.3. Mostrar números enteros en pantalla

Cuando queremos escribir un texto "tal cual", como en el ejemplo anterior, lo encerramos entre comillas. Pero no siempre querremos escribir textos prefijados. En muchos casos, se tratará de algo que habrá que calcular.

El ejemplo más sencillo es el de una operación matemática. La forma de realizarla es sencilla: no usar comillas en WriteLine. Entonces, C# intentará analizar el contenido para ver qué quiere decir. Por ejemplo, para sumar 3 y 4 bastaría hacer:

```
public class Ejemplo01suma
{
    public static void Main()
    {
        System.Console.WriteLine(3+4);
    }
}
```

Ejercicio propuesto: crea un programa que diga el resultado de sumar 118 y 56.

1.4. Operaciones aritméticas básicas

Está claro que el símbolo de la suma será un +, y podemos esperar cual será el de la resta, pero alguna de las operaciones matemáticas habituales tiene símbolos menos intuitivos. Veamos cuales son los más importantes:

Operador	Operación
+	Suma
-	Resta, negación
*	Multiplicación
/	División
%	Resto de la división ("módulo")

Ejercicios propuestos:

- Hacer un programa que calcule el producto de los números 12 y 13.
- Hacer un programa que calcule la diferencia (resta) entre 321 y 213.
- Hacer un programa que calcule el resultado de dividir 301 entre 3.
- Hacer un programa que calcule el resto de la división de 301 entre 3.

1.4.1. Orden de prioridad de los operadores

Sencillo:

- En primer lugar se realizarán las operaciones indicadas entre paréntesis.
- Luego la negación.
- Después las multiplicaciones, divisiones y el resto de la división.
- Finalmente, las sumas y las restas.
- En caso de tener igual prioridad, se analizan de izquierda a derecha.

Ejercicio propuesto: Calcular (a mano y después comprobar desde C#) el resultado de estas operaciones:

- $-2 + 3 * 5$
- $(20+5) \% 6$
- $15 + -5*6 / 10$
- $2 + 10 / 5 * 2 - 7 \% 1$

1.4.2. Introducción a los problemas de desbordamiento

El espacio del que disponemos para almacenar los números es limitado. Si el resultado de una operación es un número "demasiado grande", obtendremos un mensaje de error o un resultado erróneo. Por eso en los primeros ejemplos usaremos números pequeños. Más adelante veremos a qué se debe realmente este problema y cómo evitarlo. Como anticipo, el siguiente programa ni siquiera compila, porque el compilador sabe que el resultado va a ser "demasiado grande":

```
public class Ejemplo01multiplic
{
```

```
public static void Main()
{
    System.Console.WriteLine(10000000*10000000);
}
```

1.5. Introducción a las variables: int

Las **variables** son algo que no contiene un valor predeterminado, un espacio de memoria al que nosotros asignamos un nombre y en el que podremos almacenar datos.

El primer ejemplo nos permitía escribir "Hola". El segundo nos permitía sumar dos números que habíamos prefijado en nuestro programa. Pero esto tampoco es "lo habitual", sino que esos números dependerán de valores que haya tecleado el usuario o de cálculos anteriores.

Por eso necesitaremos usar variables, zonas de memoria en las que guardemos los datos con los que vamos a trabajar y también los resultados temporales. Como primer ejemplo, vamos a ver lo que haríamos para sumar dos números enteros que fijásemos en el programa.

1.5.1. Definición de variables: números enteros

Para usar una cierta variable primero hay que **declararla**: indicar su nombre y el tipo de datos que querremos guardar.

El primer tipo de datos que usaremos serán números enteros (sin decimales), que se indican con "int" (abreviatura del inglés "integer"). Después de esta palabra se indica el nombre que tendrá la variable:

```
int primerNumero;
```

Esa orden reserva espacio para almacenar un número entero, que podrá tomar distintos valores, y al que nos referiremos con el nombre "primerNumero".

1.5.2. Asignación de valores

Podemos darle un valor a esa variable durante el programa haciendo

```
primerNumero = 234;
```

O también podemos darles un valor inicial ("initializarlas") antes de que empiece el programa, en el mismo momento en que las definimos:

```
int primerNumero = 234;
```

O incluso podemos definir e inicializar más de una variable a la vez

```
int primerNumero = 234, segundoNumero = 567;
```

(esta línea reserva espacio para dos variables, que usaremos para almacenar números enteros; una de ellas se llama primerNumero y tiene como valor inicial 234 y la otra se llama segundoNumero y tiene como valor inicial 567).

Después ya podemos hacer operaciones con las variables, igual que las hacíamos con los números:

```
suma = primerNumero + segundoNumero;
```

1.5.3. Mostrar el valor de una variable en pantalla

Una vez que sabemos cómo mostrar un número en pantalla, es sencillo mostrar el valor de una variable. Para un número hacíamos cosas como

```
System.Console.WriteLine(3+4);
```

pero si se trata de una variable es idéntico:

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

O bien, si queremos mostrar un texto además del valor de la variable, podemos indicar el texto entre comillas, detallando con {0} en qué parte del texto queremos que aparezca el valor de la variable, de la siguiente forma:

```
System.Console.WriteLine("La suma es {0}", suma);
```

Si se trata de más de una variable, indicaremos todas ellas tras el texto, y detallaremos dónde debe aparecer cada una de ellas, usando {0}, {1} y así sucesivamente:

```
System.Console.WriteLine("La suma de {0} y {1} es {2}",
    primerNumero, segundoNumero, suma);
```

Ya sabemos todo lo suficiente para crear nuestro programa que sume dos números usando variables:

```
public class Ejemplo02
{
    public static void Main()
    {
        int primerNumero;
        int segundoNumero;
        int suma;

        primerNumero = 234;
        segundoNumero = 567;
        suma = primerNumero + segundoNumero;

        System.Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}
```

Repasemos lo que hace:

- (Nos saltamos todavía los detalles de qué quieren decir "public", "class", "static" y "void").
- `Main()` indica donde comienza el cuerpo del programa, que se delimita entre llaves { y }.
- `int primerNumero;` reserva espacio para guardar un número entero, al que llamaremos `primerNumero`.
- `int segundoNumero;` reserva espacio para guardar otro número entero, al que llamaremos `segundoNumero`.
- `int suma;` reserva espacio para guardar un tercer número entero, al que llamaremos `suma`.
- `primerNumero = 234;` da el valor del primer número que queremos sumar
- `segundoNumero = 567;` da el valor del segundo número que queremos sumar
- `suma = primerNumero + segundoNumero;` halla la suma de esos dos números y la guarda en otra variable, en vez de mostrarla directamente en pantalla.
- `System.Console.WriteLine("La suma de {0} y {1} es {2}", primerNumero, segundoNumero, suma);` muestra en pantalla el texto y los valores de las tres variables (los dos números iniciales y su suma).

Ejercicio propuesto: Hacer un programa que calcule el producto de los números 121 y 132, usando variables.

1.6. Identificadores

Estos nombres de variable (lo que se conoce como "**identificadores**") pueden estar formados por letras, números o el símbolo de subrayado (_) y deben comenzar por letra o subrayado. No deben tener espacios entre medias, y hay que recordar que las vocales acentuadas y la eñe son problemáticas, porque no son letras "estándar" en todos los idiomas.

Por eso, no son nombres de variable válidos:

1numero	(empieza por número)
un numero	(contiene un espacio)
Año1	(tiene una eñe)
MásDatos	(tiene una vocal acentuada)

Tampoco podremos usar como identificadores las **palabras reservadas** de C#. Por ejemplo, la palabra "int" se refiere a que cierta variable guardará un número entero, así que esa palabra "int" no la podremos usar tampoco como nombre de variable (pero no vamos a incluir ahora una lista de palabras reservadas de C#, ya nos iremos encontrando con ellas).

De momento, intentaremos usar nombres de variables que a nosotros nos resulten claros, y que no parezca que puedan ser alguna orden de C#.

Hay que recordar que en C# las **mayúsculas y minúsculas** se consideran diferentes, de modo que si intentamos hacer

```
PrimerNumero = 0;
primernumero = 0;
```

o cualquier variación similar, el compilador protestará y nos dirá que no conoce esa variable, porque la habíamos declarado como

```
int primerNumero;
```

1.7. Comentarios

Podemos escribir comentarios, que el compilador ignora, pero que pueden servir para aclararnos cosas a nosotros. Se escriben entre /* y */:

```
int suma; /* Porque guardaré el valor para usarlo más tarde */
```

Es conveniente escribir comentarios que aclaren la misión de las partes de nuestros programas que puedan resultar menos claras a simple vista. Incluso suele ser aconsejable que el programa comience con un comentario, que nos recuerde qué hace el programa sin que necesitemos mirarlo de arriba a abajo. Un ejemplo casi exagerado:

```
/* ---- Ejemplo en C#: sumar dos números prefijados ---- */

public class Ejemplo02b
{
    public static void Main()
    {
        int primerNumero = 234;
        int segundoNumero = 567;
        int suma; /* Guardaré el valor para usarlo más tarde */

        /* Primero calculo la suma */
        suma = primerNumero + segundoNumero;

        /* Y después muestro su valor */
        System.Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}
```

Un comentario puede empezar en una línea y terminar en otra distinta, así:

```
/* Esto
   es un comentario que
   ocupa más de una línea
*/
```

También es posible declarar otro tipo de comentarios, que comienzan con doble barra y terminan cuando se acaba la línea (estos comentarios, claramente, no podrán ocupar más de una línea). Son los "comentarios al estilo de C++":

```
// Este es un comentario "al estilo C++"
```

1.8. Datos por el usuario: `ReadLine`

Si queremos que sea el usuario de nuestro programa quien teclee los valores, necesitamos una nueva orden, que nos permita leer desde teclado. Pues bien, al igual que tenemos `System.Console.WriteLine` ("escribir línea"), también existe `System.Console.ReadLine` ("leer línea"). Para leer textos, haríamos

```
texto = System.Console.ReadLine();
```

pero eso ocurrirá en el próximo tema, cuando veamos cómo manejar textos. De momento, nosotros sólo sabemos manipular números enteros, así que deberemos convertir ese dato a un número entero, usando `Convert.ToInt32`:

```
primerNumero = System.Convert.ToInt32( System.Console.ReadLine() );
```

Un ejemplo de programa que sume dos números tecleados por el usuario sería:

```
public class Ejemplo03
{
    public static void Main()
    {
        int primerNumero;
        int segundoNumero;
        int suma;

        System.Console.WriteLine("Introduce el primer número");
        primerNumero = System.Convert.ToInt32(
            System.Console.ReadLine());
        System.Console.WriteLine("Introduce el segundo número");
        segundoNumero = System.Convert.ToInt32(
            System.Console.ReadLine());
        suma = primerNumero + segundoNumero;

        System.Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}
```

Va siendo hora de hacer una pequeña mejora: no es necesario repetir "System." al principio de la mayoría de las órdenes que tienen que ver con el sistema (por ahora, las de consola y las de conversión), si al principio del programa utilizamos "using System":

```
using System;

public class Ejemplo04
{
    public static void Main()
    {
        int primerNumero;
        int segundoNumero;
        int suma;

        Console.WriteLine("Introduce el primer número");
        primerNumero = Convert.ToInt32(Console.ReadLine());
```

```
Console.WriteLine("Introduce el segundo número");
segundoNumero = Convert.ToInt32(Console.ReadLine());
suma = primerNumero + segundoNumero;

Console.WriteLine("La suma de {0} y {1} es {2}",
    primerNumero, segundoNumero, suma);
}

}
```

Ejercicios propuestos:

1. Multiplicar dos números tecleados por usuario.
2. El usuario tecleará dos números (x e y), y el programa deberá calcular cual es el resultado de su división y el resto de esa división.
3. El usuario tecleará dos números (a y b), y el programa mostrará el resultado de la operación $(a+b)*(a-b)$ y el resultado de la operación a^2-b^2 .
4. Sumar tres números tecleados por usuario.
5. Pedir al usuario un número y mostrar su tabla de multiplicar. Por ejemplo, si el número es el 3, debería escribirse algo como

$3 \times 0 = 0$

$3 \times 1 = 3$

$3 \times 2 = 6$

...

$3 \times 10 = 30$

2. Estructuras de control

2.1. Estructuras alternativas

2.1.1. if

Vamos a ver cómo podemos comprobar si se cumplen condiciones. La primera construcción que usaremos será "**si ... entonces ...**". El formato es

```
if (condición) sentencia;
```

Vamos a verlo con un ejemplo:

```
/*-----*/
/* Ejemplo en C# nº 5: */
/* ejemplo05.cs */
/*
/* Condiciones con if */
/*
/* Introducción a C#,
/* Nacho Cabanes
/*-----*/
using System;

public class Ejemplo05
{
    public static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero>0) Console.WriteLine("El número es positivo.");
    }
}
```

Este programa pide un número al usuario. Si es positivo (mayor que 0), escribe en pantalla "El número es positivo.>"; si es negativo o cero, no hace nada.

Como se ve en el ejemplo, para comprobar si un valor numérico es mayor que otro, usamos el símbolo ">". Para ver si dos valores son iguales, usaremos dos símbolos de "igual": `if (numero==0)`. Las demás posibilidades las veremos algo más adelante. En todos los casos, la condición que queremos comprobar deberá indicarse entre paréntesis.

Este programa comienza por un comentario que nos recuerda de qué se trata. Como nuestros fuentes irán siendo cada vez más complejos, a partir de ahora incluiremos comentarios que nos permitan recordar de un vistazo qué pretendíamos hacer.

Si la orden "if" es larga, se puede partir en dos líneas para que resulte más legible:

```
if (numero>0)
    Console.WriteLine("El número es positivo.");
```

Ejercicios propuestos:

- Crear un programa que pida al usuario un número entero y diga si es par (pista: habrá que comprobar si el resto que se obtiene al dividir entre dos es cero: if ($x \% 2 == 0$) ...).
- Crear un programa que pida al usuario dos números enteros y diga cual es el mayor de ellos.
- Crear un programa que pida al usuario dos números enteros y diga si el primero es múltiplo del segundo (pista: igual que antes, habrá que ver si el resto de la división es cero: $a \% b == 0$).

2.1.2. if y sentencias compuestas

Habíamos dicho que el formato básico de "if" es `if (condición) sentencia;` Esa "sentencia" que se ejecuta si se cumple la condición puede ser una sentencia simple o una compuesta. Las sentencias **compuestas** se forman agrupando varias sentencias simples entre llaves (`{ y }`), como en este ejemplo:

```
/*
 *----- Ejemplo en C# nº 6: -----
 /* ejemplo06.cs */
 /*
 /* Condiciones con if (2)
 /* Sentencias compuestas */
 /*
 /* Introducción a C#,
 /* Nacho Cabanes
 /*-----*/
```

```
using System;

public class Ejemplo06
{
    public static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero > 0)
        {
            Console.WriteLine("El número es positivo.");
            Console.WriteLine("Recuerde que también puede usar negativos.");
        } /* Acaba el "if" */
        /* Acaba "Main" */
    } /* Acaba "Ejemplo06" */
```

En este caso, si el número es positivo, se hacen dos cosas: escribir un texto y luego... escribir otro! (Claramente, en este ejemplo, esos dos "WriteLine" podrían ser uno solo, en el que los

dos textos estuvieran separados por un "\n"; más adelante iremos encontrando casos en lo que necesitemos hacer cosas "más serias" dentro de una sentencia compuesta).

Como se ve en este ejemplo, cada nuevo "bloque" se suele escribir un poco más a la derecha que los anteriores, para que sea fácil ver dónde comienza y termina cada sección de un programa. Por ejemplo, el contenido de "Ejemplo06" está un poco más a la derecha que la cabecera "public class Ejemplo06", y el contenido de "Main" algo más a la derecha, y la sentencia compuesta que se debe realizar si se cumple la condición del "if" está algo más a la derecha que la orden "if". Este **sangrado** del texto se suele llamar **"escritura indentada"**. Un tamaño habitual para el sangrado es de 4 espacios, aunque en este texto muchas veces usaremos sólo dos espacios, para no llegar al margen derecho del papel con demasiada facilidad.

2.1.3. Operadores relacionales: <, <=, >, >=, ==, !=

Hemos visto que el símbolo ">" es el que se usa para comprobar si un número es mayor que otro. El símbolo de "menor que" también es sencillo, pero los demás son un poco menos evidentes, así que vamos a verlos:

Operador	Operación
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
==	Igual a
!=	No igual a (distinto de)

Así, un ejemplo, que diga si un número NO ES cero sería:

```
/*
/*----- Ejemplo en C# nº 7: -----
/* ejemplo07.cs
/*
/*
/* Condiciones con if (3)
/*
/*
/* Introducción a C#
/* Nacho Cabanes
/*
/*-----*/



using System;

public class Ejemplo07
{
    public static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero != 0)
            Console.WriteLine("El número no es cero.");
    }
}
```

Ejercicios propuestos:

- Crear un programa que multiplique dos números enteros de la siguiente forma: pedirá al usuario un primer número entero. Si el número que se teclee es 0, escribirá en pantalla "El producto de 0 por cualquier número es 0". Si se ha tecleado un número distinto de cero, se pedirá al usuario un segundo número y se mostrará el producto de ambos.
- Crear un programa que pida al usuario dos números reales. Si el segundo no es cero, mostrará el resultado de dividir entre el primero y el segundo. Por el contrario, si el segundo número es cero, escribirá "Error: No se puede dividir entre cero".

2.1.4. if-else

Podemos indicar lo que queremos que ocurra en caso de que no se cumpla la condición, usando la orden "else" (en caso contrario), así:

```
/*
/* Ejemplo en C# nº 8: */
/* ejemplo08.cs */
/*
/*
/* Condiciones con if (4) */
/*
/*
/* Introduccion a C#,
/* Nacho Cabanes */
/*
*/
using System;

public class Ejemplo08
{
    public static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero > 0)
            Console.WriteLine("El número es positivo.");
        else
            Console.WriteLine("El número es cero o negativo.");
    }
}
```

Podríamos intentar evitar el uso de "else" si utilizamos un "if" a continuación de otro, así:

```
/*
/* Ejemplo en C# nº 9: */
/* ejemplo09.cs */
/*
/*
/* Condiciones con if (5) */
/*
/*
/* Introduccion a C#,
/* Nacho Cabanes */
/*
*/
using System;
```

```

public class Ejemplo09
{
    public static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());

        if (numero > 0)
            Console.WriteLine("El número es positivo.");

        if (numero <= 0)
            Console.WriteLine("El número es cero o negativo.");
    }
}

```

Pero el comportamiento no es el mismo: en el primer caso (ejemplo 8) se mira si el valor es positivo; si no lo es, se pasa a la segunda orden, pero si lo es, el programa ya ha terminado. En el segundo caso (ejemplo 9), aunque el número sea positivo, se vuelve a realizar la segunda comprobación para ver si es negativo o cero, por lo que el programa es algo más lento.

Podemos enlazar varios "if" usando "else", para decir "si no se cumple esta condición, mira a ver si se cumple esta otra":

```

/*-----*/
/* Ejemplo en C# nº 10:      */
/* ejemplo10.cs               */
/* */
/* Condiciones con if (6)   */
/* */
/* Introducción a C#,        */
/* Nacho Cabanes             */
/*-----*/
using System;

public class Ejemplo10
{
    public static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());

        if (numero > 0)
            Console.WriteLine("El número es positivo.");
        else
            if (numero < 0)
                Console.WriteLine("El número es negativo.");
            else
                Console.WriteLine("El número es cero.");
    }
}

```

Ejercicio propuesto:

- Mejorar la solución a los dos ejercicios del apartado anterior, usando "else".

2.1.5. Operadores lógicos: &&, ||, !

Estas condiciones se puede **encadenar** con "y", "o", etc., que se indican de la siguiente forma

Operador	Significado
&&	Y
	O
!	No

De modo que podremos escribir cosas como

```
if ((opcion==1) && (usuario==2)) ...
if ((opcion==1) || (opcion==3)) ...
if (!(opcion==opcCorrecta)) || (tecla==ESC)) ...
```

Ejercicios propuestos:

- Crear un programa que pida una letra al usuario y diga si se trata de una vocal.
- Crear un programa que pida al usuario dos números enteros y diga "Uno de los números es positivo", "Los dos números son positivos" o bien "Ninguno de los números es positivo", según corresponda.
- Crear un programa que pida al usuario tres números reales y muestre cuál es el mayor de los tres.
- Crear un programa que pida al usuario dos números enteros cortos y diga si son iguales o, en caso contrario, cuál es el mayor de ellos.
- Crear un programa que pida al usuario su nombre, y le diga "Hola" si se llama "Juan", o bien le diga "No te conozco" si teclea otro nombre.

2.1.6. El peligro de la asignación en un "if"

Cuidado con el operador de **igualdad**: hay que recordar que el formato es `if (a==b) ...`. Si no nos acordamos y escribimos `if (a=b)`, estamos intentando asignar a "a" el valor de "b".

En algunos compiladores de lenguaje C, esto podría ser un problema serio, porque se considera válido hacer una asignación dentro de un "if" (aunque la mayoría de compiladores modernos nos avisarían de que quizás estemos asignando un valor sin pretenderlo, pero no es un "error" sino un "aviso", lo que permite que se genere un ejecutable, y podríamos pasar por alto el aviso, dando lugar a un funcionamiento incorrecto de nuestro programa).

En el caso del lenguaje C#, este riesgo no existe, porque la "condición" debe ser algo cuyo resultado "verdadero" o "falso" (un dato de tipo "bool"), de modo que obtendríamos un error de compilación "Cannot implicitly convert type 'int' to 'bool'" (*no puedo convertir un "int" a "bool"*). Es el caso del siguiente programa:

```
/*-----*/
/* Ejemplo en C# nº 11:      */
/* ejemplo11.cs               */
/*-----*/
```

```

/*
/* Condiciones con if (7) */
/* comparacion incorrecta */
/*
/* Introduccion a C#, */
/* Nacho Cabanes */
/*-----*/
using System;

public class Ejemplo11
{
    public static void Main()
    {
        int numero;

        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero == 0)
            Console.WriteLine("El número es cero.");
        else
            if (numero < 0)
                Console.WriteLine("El número es negativo.");
            else
                Console.WriteLine("El número es positivo.");
    }
}

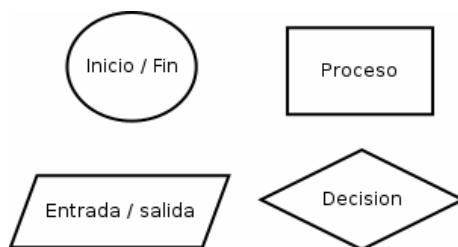
```

2.1.7. Introducción a los diagramas de flujo

A veces puede resultar difícil ver claro donde usar un "else" o qué instrucciones de las que siguen a un "if" deben ir entre llaves y cuales no. Generalmente la dificultad está en el hecho de intentar teclear directamente un programa en C#, en vez de pensar en el problema que se pretende resolver.

Para ayudarnos a centrarnos en el problema, existen notaciones gráficas, como los diagramas de flujo, que nos permiten ver mejor qué se debe hacer y cuando.

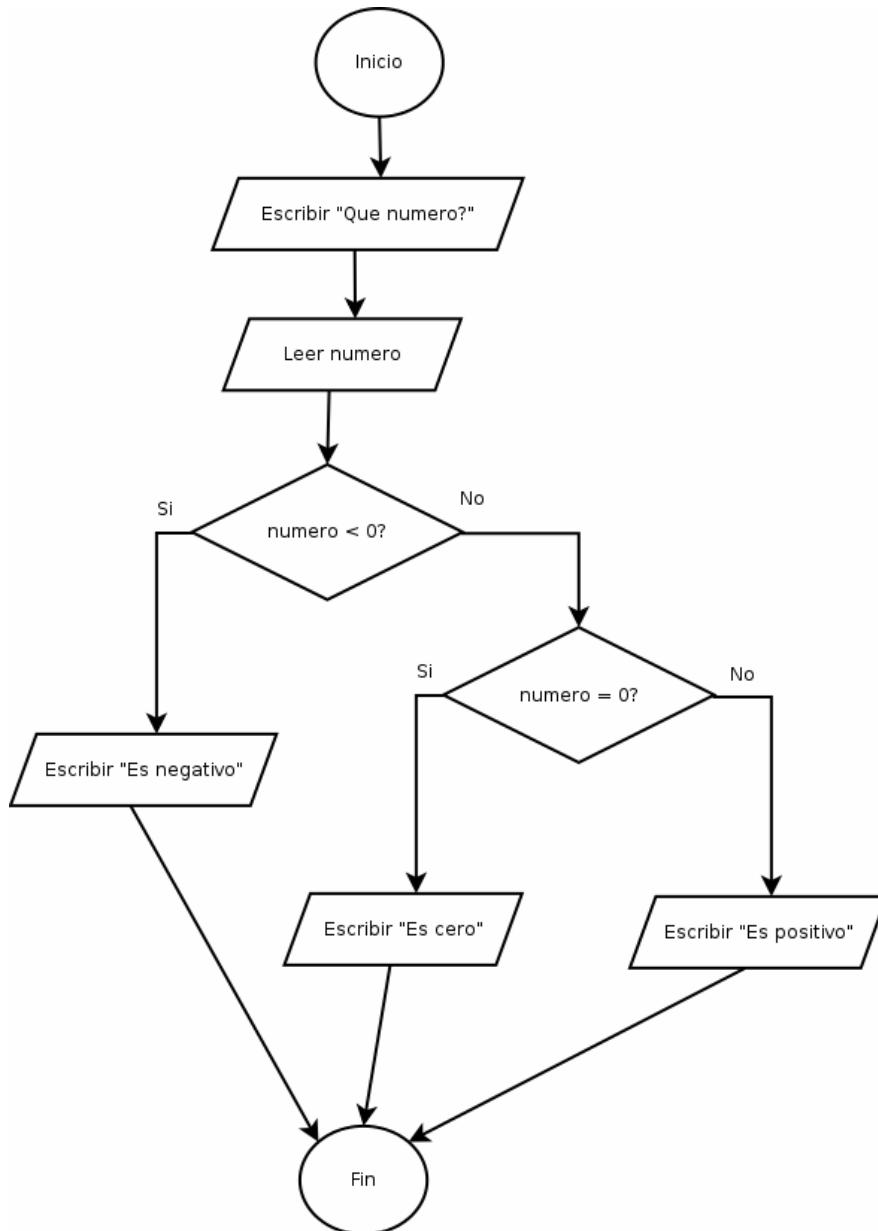
En primer lugar, vamos a ver los 4 elementos básicos de un diagrama de flujo, y luego los aplicaremos a un caso concreto.



El inicio o el final del programa se indica dentro de un círculo. Los procesos internos, como realizar operaciones, se encuadran en un rectángulo. Las entradas y salidas (escrituras en pantalla y lecturas de teclado) se indican con un paralelogramo que tenga su lado superior e

inferior horizontales, pero no tenga verticales los otros dos. Las decisiones se indican dentro de un rombo.

Vamos a aplicarlo al ejemplo de un programa que pida un número al usuario y diga si es positivo, negativo o cero:



El paso de aquí al correspondiente programa en lenguaje C# (el que vimos en el ejemplo 11) debe ser casi inmediato: sabemos como leer de teclado, como escribir en pantalla, y las decisiones serán un "if", que si se cumple ejecutará la sentencia que aparece en su salida "si" y si no se cumple ("else") ejecutará lo que aparezca en su salida "no".

Ejercicios propuestos:

- Crear el diagrama de flujo y la versión en C# de un programa que dé al usuario tres oportunidades para adivinar un número del 1 al 10.

- Crear el diagrama de flujo para el programa que pide al usuario dos números y dice si uno de ellos es positivo, si lo son los dos o si no lo es ninguno.
- Crear el diagrama de flujo para el programa que pide tres números al usuario y dice cuál es el mayor de los tres.

2.1.8. Operador condicional: ?

En C# hay otra forma de asignar un valor según se cumpla una condición o no. Es el "**operador condicional**" ?: que se usa

```
nombreVariable = condicion ? valor1 : valor2;
```

y equivale a decir "si se cumple la condición, toma el valor *valor1*; si no, toma el valor *valor2*". Un ejemplo de cómo podríamos usarlo sería para calcular el mayor de dos números:

```
numeroMayor = a>b ? a : b;
```

esto equivale a la siguiente orden "if":

```
if ( a > b )
    numeroMayor = a;
else
    numeroMayor = b;
```

Aplicado a un programa sencillo, podría ser

```
/*
/* Ejemplo en C# nº 12: */
/* ejemplo12.cs */
/*
/* El operador condicional */
/*
/* Introducción a C#, */
/* Nacho Cabanes */
/*
using System;

public class Ejemplo12
{
    public static void Main()
    {
        int a, b, mayor;

        Console.Write("Escriba un número: ");
        a = Convert.ToInt32(Console.ReadLine());
        Console.Write("Escriba otro: ");
        b = Convert.ToInt32(Console.ReadLine());

        mayor = (a>b) ? a : b;

        Console.WriteLine("El mayor de los números es {0}.", mayor);
    }
}
```

(La orden Console.Write, empleada en el ejemplo anterior, escribe un texto sin avanzar a la línea siguiente, de modo que el próximo texto que escribamos –o introduzcamos- quedará a continuación de éste).

Un segundo ejemplo, que sume o reste dos números según la opción que se escoja, sería:

```
/*-----*/
/* Ejemplo en C# nº 13:      */
/* ejemplo13.cs              */
/*                         */
/* Operador condicional - 2 */
/*                         */
/* Introducción a C#,        */
/* Nacho Cabanes             */
/*-----*/
using System;

public class Ejemplo13
{
    public static void Main()
    {
        int a, b, operacion, resultado;

        Console.WriteLine("Escriba un número: ");
        a = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("Escriba otro: ");
        b = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("Escriba una operación (1 = resta; otro = suma): ");
        operacion = Convert.ToInt32(Console.ReadLine());

        resultado = (operacion == 1) ? a-b : a+b;
        Console.WriteLine("El resultado es {0}.\n", resultado);
    }
}
```

Ejercicios propuestos:

- Crear un programa que use el operador condicional para mostrar un el valor absoluto de un número de la siguiente forma: si el número es positivo, se mostrará tal cual; si es negativo, se mostrará cambiado de signo.
- Usar el operador condicional para calcular el menor de dos números.

2.1.10. switch

Si queremos ver **varios posibles valores**, sería muy pesado tener que hacerlo con muchos "if" seguidos o encadenados. La alternativa es la orden "switch", cuya sintaxis es

```
switch (expresión)
{
    case valor1: sentencial;
        break;
    case valor2: sentencia2;
```

```

        sentencia2b;
        break;
    ...
    case valorN: sentenciaN;
        break;
    default:
        otraSentencia;
        break;
}

```

Es decir, se escribe tras "switch" la expresión a analizar, entre paréntesis. Después, tras varias órdenes "case" se indica cada uno de los valores posibles. Los pasos (porque pueden ser varios) que se deben dar si se trata de ese valor se indican a continuación, terminando con "break". Si hay que hacer algo en caso de que no se cumpla ninguna de las condiciones, se detalla después de la palabra "default". Si dos casos tienen que hacer lo mismo, se añade "goto case" a uno de ellos para indicarlo.

Vamos con un ejemplo, que diga si el símbolo que introduce el usuario es una cifra numérica, un espacio u otro símbolo. Para ello usaremos un dato de tipo "char" (carácter), que veremos con más detalle en el próximo tema. De momento nos basta que deberemos usar Convert.ToChar si lo leemos desde teclado con ReadLine, y que le podemos dar un valor (o compararlo) usando comillas simples:

```

/*-----*/
/* Ejemplo en C# nº 14:      */
/* ejemplo14.cs               */
/*                           */
/* La orden "switch"  (1)   */
/*                           */
/*-----*/
/* Introduccion a C#,       */
/* Nacho Cabanes            */
/*-----*/
using System;

public class Ejemplo14
{
    public static void Main()
    {
        char letra;

        Console.WriteLine("Introduce una letra");
        letra = Convert.ToChar( Console.ReadLine() );

        switch (letra)
        {
            case ' ': Console.WriteLine("Espacio.");
                        break;
            case '1': goto case '0';
            case '2': goto case '0';
            case '3': goto case '0';
            case '4': goto case '0';
            case '5': goto case '0';
            case '6': goto case '0';
            case '7': goto case '0';
            case '8': goto case '0';
        }
    }
}

```

```
        case '9': goto case '0';
    case '0': Console.WriteLine("Dígito.");
                break;
    default: Console.WriteLine("Ni espacio ni dígito.");
                break;
}
}
```

Cuidado quien venga del lenguaje C: en C se puede dejar que un caso sea manejado por el siguiente, lo que se consigue si no se usa "break", mientras que C# siempre obliga a usar "break" o "goto" al final de cada caso, con la única excepción de que un caso no haga absolutamente nada que no sea dejar pasar el control al siguiente caso, y en ese caso se puede dejar totalmente vacío:

```
/*-----*/
/* Ejemplo en C# nº 14b:      */
/* ejemplo14b.cs               */
/*                               */
/* La orden "switch"  (1b)    */
/*                               */
/* Introduccion a C#,          */
/* Nacho Cabanes              */
/*-----*/
using System;

public class Ejemplo14b
{
    public static void Main()
    {
        char letra;

        Console.WriteLine("Introduce una letra");
        letra = Convert.ToChar( Console.ReadLine() );

        switch (letra)
        {
            case ' ': Console.WriteLine("Espacio.");
                        break;
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
            case '0': Console.WriteLine("Dígito.");
                        break;
            default:  Console.WriteLine("Ni espacio ni dígito.");
                        break;
        }
    }
}
```

En el lenguaje C, que es más antiguo, sólo se podía usar "switch" para comprobar valores de variables "simples" (numéricas y caracteres); en C#, que es un lenguaje más evolucionado, se puede usar también para comprobar valores de cadenas de texto ("strings").

Una cadena de texto, como veremos con más detalle en el próximo tema, se declara con la palabra "string", se puede leer de teclado con ReadLine (sin necesidad de convertir) y se le puede dar un valor desde programa si se indica entre comillas dobles. Por ejemplo, un programa que nos salude de forma personalizada si somos "Juan" o "Pedro" podría ser:

```
/*
 * Ejemplo en C# nº 15:
 * ejemplo15.cs
 */
/* La orden "switch" (2)
 */
/* Introduccion a C#,
 * Nacho Cabanes
 */
using System;

public class Ejemplo15
{
    public static void Main()
    {
        string nombre;

        Console.WriteLine("Introduce tu nombre");
        nombre = Console.ReadLine();

        switch (nombre)
        {
            case "Juan": Console.WriteLine("Bienvenido, Juan.");
                           break;
            case "Pedro": Console.WriteLine("Que tal estas, Pedro.");
                           break;
            default:   Console.WriteLine("Procede con cautela, desconocido.");
                           break;
        }
    }
}
```

Ejercicios propuestos:

- Crear un programa que lea una letra tecleada por el usuario y diga si se trata de una vocal, una cifra numérica o una consonante (pista: habrá que usar un dato de tipo "char").
- Crear un programa que lea una letra tecleada por el usuario y diga si se trata de un signo de puntuación, una cifra numérica o algún otro carácter.
- Repetir los dos programas anteriores, empleando "if" en lugar de "switch".

2.2. Estructuras repetitivas

Hemos visto cómo comprobar condiciones, pero no cómo hacer que una cierta parte de un programa se repita un cierto número de veces o mientras se cumpla una condición (lo que llamaremos un "**bucle**"). En C# tenemos varias formas de conseguirlo.

2.2.1. while

Si queremos hacer que una sección de nuestro programa se repita mientras se cumpla una cierta condición, usaremos la orden "while". Esta orden tiene dos formatos distintos, según comprobemos la condición al principio o al final.

En el primer caso, su sintaxis es

```
while (condición)
    sentencia;
```

Es decir, la sentencia se repetirá **mientras** la condición sea cierta. Si la condición es falsa ya desde un principio, la sentencia no se ejecuta nunca. Si queremos que se repita más de una sentencia, basta agruparlas entre { y }.

Un ejemplo que nos diga si cada número que tecleemos es positivo o negativo, y que pare cuando tecleemos el número 0, podría ser:

```
/*-----
/* Ejemplo en C# nº 16:      */
/* ejemplo16.cs               */
/*                               */
/* La orden "while"           */
/*                               */
/* Introducción a C#,          */
/* Nacho Cabanes              */
/*-----*/
using System;

public class Ejemplo16
{
    public static void Main()
    {
        int numero;

        Console.WriteLine("Teclea un número (0 para salir): ");
        numero = Convert.ToInt32(Console.ReadLine());

        while (numero != 0)
        {
            if (numero > 0) Console.WriteLine("Es positivo");
            else Console.WriteLine("Es negativo");

            Console.WriteLine("Teclea otro número (0 para salir): ");
            numero = Convert.ToInt32(Console.ReadLine());
        }
    }
}
```

```
    }
```

En este ejemplo, si se introduce 0 la primera vez, la condición es falsa y ni siquiera se entra al bloque del "while", terminando el programa inmediatamente.

Ejercicios propuestos:

- Crear un programa que pida al usuario su contraseña. Deberá terminar cuando introduzca como contraseña la palabra "clave", pero volvérsele a pedir tantas veces como sea necesario.
- Crea un programa que escriba en pantalla los números del 1 al 10, usando "while".
- Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "while".
- Crear un programa calcule cuantas cifras tiene un número entero positivo (pista: se puede hacer dividiendo varias veces entre 10).

2.2.2. do ... while

Este es el otro formato que puede tener la orden "while": la condición se comprueba **al final**. El punto en que comienza a repetirse se indica con la orden "do", así:

```
do
    sentencia;
while (condición)
```

Al igual que en el caso anterior, si queremos que se repitan varias órdenes (es lo habitual), deberemos encerrarlas entre llaves.

Como ejemplo, vamos a ver cómo sería el típico programa que nos pide una clave de acceso y no nos deja entrar hasta que tecleemos la clave correcta:

```
/*
 * Ejemplo en C# nº 17:
 * ejemplo17.cs
 */
/*
 * La orden "do..while"
 */
/*
 * Introducción a C#,
 * Nacho Cabanes
 */
```

```
using System;

public class Ejemplo17
{
    public static void Main()
    {

        int valida = 711;
        int clave;

        do
        {
```

```

Console.Write("Introduzca su clave numérica: ");
clave = Convert.ToInt32(Console.ReadLine());

if (clave != valida)
    Console.WriteLine("No válida!\n");

}

while (clave != valida);

Console.WriteLine("Aceptada.\n");

}

```

En este caso, se comprueba la condición al final, de modo que se nos preguntará la clave al menos una vez. Mientras que la respuesta que demos no sea la correcta, se nos vuelve a preguntar. Finalmente, cuando tecleamos la clave correcta, el ordenador escribe "Aceptada" y termina el programa.

Si preferimos que la clave sea un texto en vez de un número, los cambios al programa son mínimos, basta con usar "string":

```

/*-----*/
/* Ejemplo en C# nº 18:          */
/* ejemplo18.cs                  */
/*                               */
/* La orden "do..while" (2)      */
/*                               */
/* Introducción a C#,           */
/* Nacho Cabanes                */
/*-----*/

using System;

public class Ejemplo18
{
    public static void Main()
    {

        string valida = "secreto";
        string clave;

        do
        {
            Console.Write("Introduzca su clave: ");
            clave = Console.ReadLine();

            if (clave != valida)
                Console.WriteLine("No válida!\n");

        }
        while (clave != valida);

        Console.WriteLine("Aceptada.\n");
    }
}

```

```
}
```

Ejercicios propuestos:

- Crear un programa que pida números positivos al usuario, y vaya calculando la suma de todos ellos (terminará cuando se teclea un número negativo o cero).
- Crea un programa que escriba en pantalla los números del 1 al 10, usando "do..while".
- Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "do..while".
- Crea un programa que pida al usuario su nombre y su contraseña, y no le permita seguir hasta que introduzca como nombre "Pedro" y como contraseña "Peter".

2.2.3. for

Ésta es la orden que usaremos habitualmente para crear partes del programa que **se repitan** un cierto número de veces. El formato de "for" es

```
for (valorInicial; CondiciónRepetición; Incremento)
    Sentencia;
```

Así, para **contar del 1 al 10**, tendríamos 1 como valor inicial, ≤ 10 como condición de repetición, y el incremento sería de 1 en 1. Es muy habitual usar la letra "i" como contador, cuando se trata de tareas muy sencillas, así que el valor inicial sería "i=1", la condición de repetición sería "i ≤ 10 " y el incremento sería "i=i+1":

```
for (i=1; i<=10; i=i+1)
    ...

```

La orden para incrementar el valor de una variable ("i = i+1") se puede escribir de la forma abreviada "i++", como veremos con más detalle en el próximo tema.

En general, será preferible usar nombres de variable más descriptivos que "i". Así, un programa que escribiera los números del 1 al 10 podría ser:

```
/*
/* Ejemplo en C# nº 19: */
/* ejemplo19.cs */
/*
/* Uso básico de "for" */
/*
/* Introducción a C#,
/* Nacho Cabanes
/*
*/
```

```
using System;

public class Ejemplo19
{
    public static void Main()
    {
        int contador;
```

```

for (contador=1; contador<=10; contador++)
    Console.WriteLine("{0} ", contador);

}
}

```

Ejercicios propuestos:

- Crear un programa que muestre los números del 15 al 5, descendiendo (pista: en cada pasada habrá que descontar 1, por ejemplo haciendo `i=i-1`, que se puede abreviar `i--`).
- Crear un programa que muestre los primeros ocho números pares (pista: en cada pasada habrá que aumentar de 2 en 2, o bien mostrar el doble del valor que hace de contador).

En un "for", realmente, la parte que hemos llamado "Incremento" no tiene por qué incrementar la variable, aunque ése es su uso más habitual. Es simplemente una orden que se ejecuta cuando se termine la "Sentencia" y antes de volver a comprobar si todavía se cumple la condición de repetición.

Por eso, si escribimos la siguiente línea:

```
for (contador=1; contador<=10; )
```

la variable "contador" no se incrementa nunca, por lo que nunca se cumplirá la condición de salida: nos quedamos encerrados dando vueltas dentro de la orden que siga al "for". El programa no termina nunca. Se trata de un "bucle sin fin".

Un caso todavía más exagerado de algo a lo que se entra y de lo que no se sale sería la siguiente orden:

```
for ( ; ; )
```

Los bucles "for" se pueden **anidar** (incluir uno dentro de otro), de modo que podríamos escribir las tablas de multiplicar del 1 al 5 con:

```

/*
/* Ejemplo en C# nº 20: */
/* ejemplo20.cs */
/*
/* "for" anidados */
/*
/* Introduccion a C#, */
/* Nacho Cabanes */
/*
*/

```

```

using System;

public class Ejemplo20
{
    public static void Main()
    {

```

```

int tabla, numero;

for (tabla=1; tabla<=5; tabla++)
{
    for (numero=1; numero<=10; numero++)
        Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                          tabla*numero);
}

```

En estos ejemplos que hemos visto, después de "for" había una única sentencia. Si queremos que se hagan varias cosas, basta definirlas como un **bloque** (una sentencia compuesta) encerrándolas entre llaves. Por ejemplo, si queremos mejorar el ejemplo anterior haciendo que deje una línea en blanco entre tabla y tabla, sería:

```

/*
 * Ejemplo en C# nº 21:
 * ejemplo21.cs
 */
/* "for" anidados (2)
 */
/* Introduccion a C#,
 * Nacho Cabanes
 */

using System;

public class Ejemplo21
{
    public static void Main()
    {
        int tabla, numero;

        for (tabla=1; tabla<=5; tabla++)
        {
            for (numero=1; numero<=10; numero++)
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                                  tabla*numero);

            Console.WriteLine();
        }
    }
}

```

Para "contar" no necesariamente hay que usar **números**. Por ejemplo, podemos contar con letras así:

```
/*
/* Ejemplo en C# nº 22: */
/* ejemplo22.cs */
/*
/* "for" que usa "char" */
/*
/* Introduccion a C#, */
/* Nacho Cabanes */
/*
using System;

public class Ejemplo22
{
    public static void Main()
    {

        char letra;

        for (letra='a'; letra<='z'; letra++)
            Console.WriteLine("{0} ", letra);

    }
}
```

En este caso, empezamos en la "a" y terminamos en la "z", aumentando de uno en uno.

Si queremos contar de forma **decreciente**, o de dos en dos, o como nos interese, basta indicarlo en la condición de finalización del "for" y en la parte que lo incrementa:

```
/*
/* Ejemplo en C# nº 23: */
/* ejemplo23.cs */
/*
/* "for" que descuenta */
/*
/* Introduccion a C#, */
/* Nacho Cabanes */
/*
using System;

public class Ejemplo23
{
    public static void Main()
    {

        char letra;

        for (letra='z'; letra>='a'; letra--)
            Console.WriteLine("{0} ", letra);

    }
}
```

Ejercicios propuestos:

- Crear un programa que muestre las letras de la Z (mayúscula) a la A (mayúscula, descendiendo).
- Crear un programa que escriba en pantalla la tabla de multiplicar del 5.
- Crear un programa que escriba en pantalla los números del 1 al 50 que sean múltiplos de 3 (pista: habrá que recorrer todos esos números y ver si el resto de la división entre 3 resulta 0).

2.3. Sentencia break: termina el bucle

Podemos salir de un bucle "for" antes de tiempo con la orden "**break**":

```
/*
 * Ejemplo en C# nº 24:
 */
/* ejemplo24.cs
 */
/* "for" interrumpido con
 */
/* "break"
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes
 */
/*-----*/
```

```
using System;

public class Ejemplo24
{
    public static void Main()
    {
        int contador;

        for (contador=1; contador<=10; contador++)
        {
            if (contador==5)
                break;

            Console.WriteLine("{0} ", contador);
        }
    }
}
```

El resultado de este programa es:

1 2 3 4

(en cuanto se llega al valor 5, se interrumpe el "for", por lo que no se alcanza el valor 10).

2.4. Sentencia continue: fuerza la siguiente iteración

Podemos saltar alguna repetición de un bucle con la orden "**continue**":

```
/*
/* Ejemplo en C# nº 25: */
/* ejemplo25.cs */
/*
/*
/* "for" interrumpido con */
/* "continue" */
/*
/* Introduccion a C#, */
/* Nacho Cabanes */
/*
using System;

public class Ejemplo25
{
    public static void Main()
    {

        int contador;

        for (contador=1; contador<=10; contador++)
        {
            if (contador==5)
                continue;

            Console.Write("{0} ", contador);
        }
    }
}
```

El resultado de este programa es:

1 2 3 4 6 7 8 9 10

En él podemos observar que no aparece el valor 5.

Ejercicios resueltos:

- ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; i<4; i++) Console.Write("{0} ",i);
```

Respuesta: los números del 1 al 3 (se empieza en 1 y se repite mientras sea menor que 4).

- ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; i>4; i++) Console.Write("{0} ",i);
```

Respuesta: no escribiría nada, porque la condición es falsa desde el principio.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i=1; i<=4; i++ ); Console.WriteLine("{0} ",i);
```

Respuesta: escribe un 5, porque hay un punto y coma después del "for", de modo que repite cuatro veces una orden vacía, y cuando termina, "i" ya tiene el valor 5.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i=1; i<4; ) Console.WriteLine("{0} ",i);
```

Respuesta: escribe "1" continuamente, porque no aumentamos el valor de "i", luego nunca se llegará a cumplir la condición de salida.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i=1; ; i++ ) Console.WriteLine("{0} ",i);
```

Respuesta: escribe números crecientes continuamente, comenzando en uno y aumentando una unidad en cada pasada, pero sin terminar.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i= 0 ; i<= 4 ; i++ ) {
    if ( i == 2 ) continue ;
    Console.WriteLine("{0} ",i);
}
```

Respuesta: escribe los números del 0 al 4, excepto el 2.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i= 0 ; i<= 4 ; i++ ) {
    if ( i == 2 ) break ;
    Console.WriteLine("{0} ",i);
}
```

Respuesta: escribe los números 0 y 1 (interrumpe en el 2).

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i= 0 ; i<= 4 ; i++ ) {
    if ( i == 10 ) continue ;
    Console.WriteLine("{0} ",i);
```

```
}
```

Respuesta: escribe los números del 0 al 4, porque la condición del "continue" nunca se llega a dar.

- ¿Qué escribiría en pantalla este fragmento de código?

```
for ( i= 0 ; i<= 4 ; i++)
    if ( i == 2 ) continue ;
    Console.WriteLine("{0} ",i);
```

Respuesta: escribe 5, porque no hay llaves tras el "for", luego sólo se repite la orden "if".

2.5. Sentencia goto

El lenguaje C# también permite la orden "**goto**", para hacer saltos incondicionales. Su uso indisciplinado está muy mal visto, porque puede ayudar a hacer programas llenos de saltos, difíciles de seguir. Pero en casos concretos puede ser muy útil, por ejemplo, para salir de un bucle muy anidado (un "for" dentro de otro "for" que a su vez está dentro de otro "for": en este caso, "break" sólo saldría del "for" más interno).

El formato de "goto" es

```
goto donde;
```

y la posición de salto se indica con su nombre seguido de dos puntos (:)

donde:

como en el siguiente ejemplo:

```
/*-----*/
/* Ejemplo en C# nº 26: */
/* ejemplo26.cs */
/*
/* "for" y "goto"
/*
/* Introducción a C#,
/* Nacho Cabanes
/*-----*/
```

```
using System;
```

```
public class Ejemplo26
{
    public static void Main()
    {
```

```
        int i, j;
```

```
        for (i=0; i<=5; i++)
```

```

for (j=0; j<=20; j+=2)
{
    if ((i==1) && (j>=7))
        goto salida;
    Console.WriteLine("i vale {0} y j vale {1}.", i, j);
}

salida:
Console.Write("Fin del programa");

}

```

El resultado de este programa es:

```

i vale 0 y j vale 0.
i vale 0 y j vale 2.
i vale 0 y j vale 4.
i vale 0 y j vale 6.
i vale 0 y j vale 8.
i vale 0 y j vale 10.
i vale 0 y j vale 12.
i vale 0 y j vale 14.
i vale 0 y j vale 16.
i vale 0 y j vale 18.
i vale 0 y j vale 20.
i vale 1 y j vale 0.
i vale 1 y j vale 2.
i vale 1 y j vale 4.
i vale 1 y j vale 6.
Fin del programa

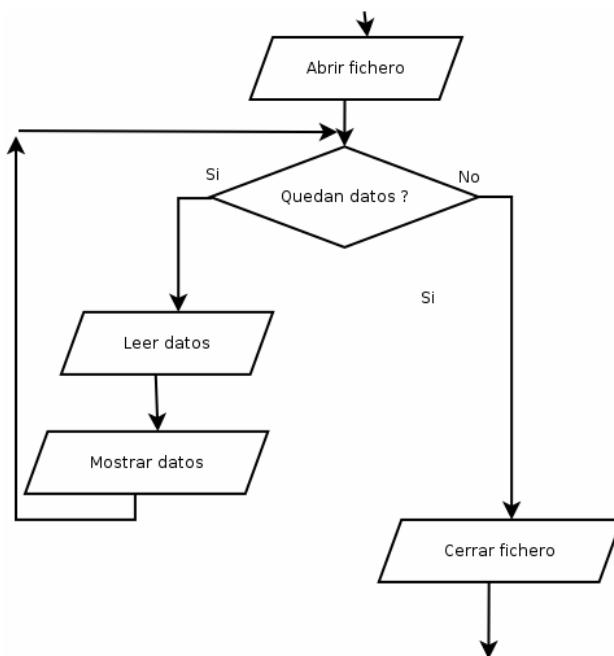
```

Vemos que cuando $i=1$ y $j \geq 7$, se sale de los dos "for".

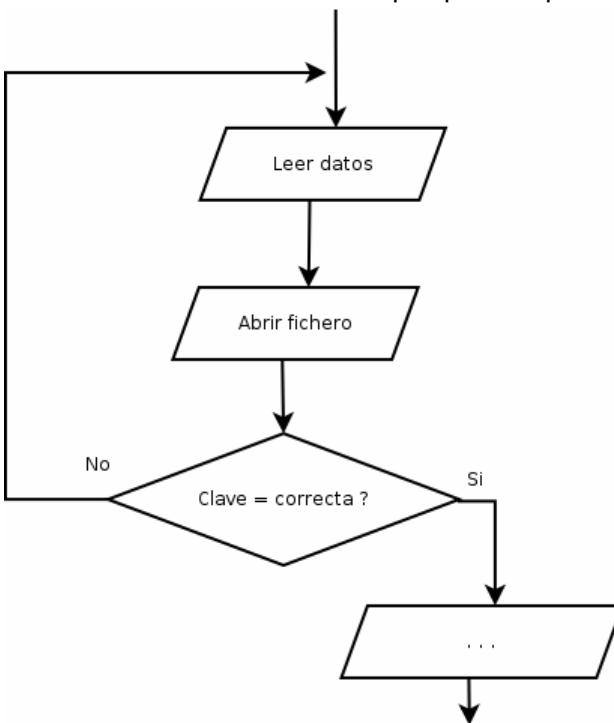
2.6. Más sobre diagramas de flujo. Diagramas de Chapin.

Cuando comenzamos el tema, vimos cómo ayudarnos de los diagramas de flujo para plantear lo que un programa debe hacer. Si entendemos esta herramienta, el paso a C (o a casi cualquier otro lenguaje de programación es sencillo). Pero este tipo de diagramas es antiguo, no tiene en cuenta todas las posibilidades del lenguaje C (y de muchos otros lenguajes actuales). Por ejemplo, no existe una forma clara de representar una orden "switch", que equivaldría a varias condiciones encadenadas.

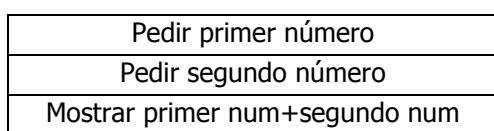
Por su parte, un bucle "while" se vería como una condición que hace que algo se repita (una flecha que vuelve hacia atrás, al punto en el que se comprobaba la condición):



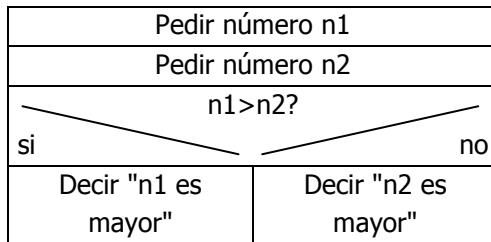
Y un "do..while" como una condición al final de un bloque que se repite:



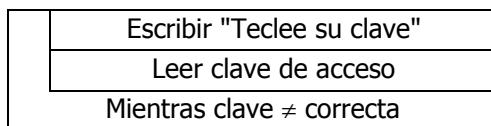
Aun así, existen otras notaciones más modernas y que pueden resultar más cómodas. Sólo comentaremos una: los diagramas de Chapin. En ellos se representa cada orden dentro de una caja:



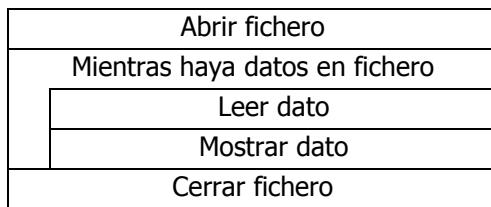
Las condiciones se indican dividiendo las cajas en dos:



Y las condiciones repetitivas se indican dejando una barra a la izquierda, que marca qué es lo que se repite, tanto si la condición se comprueba al final (do..while):



como si se comprueba al principio (while):



En ambos casos, no existe una gráfica "clara" para los "for".

2.7. El caso de "foreach"

Nos queda por ver otra orden que permite hacer cosas repetitivas: "foreach" (se traduciría "para cada"). La veremos más adelante, cuando manejemos estructuras de datos más complejas, que es en las que la nos resultará útil.

2.8. Recomendación de uso para los distintos tipos de bucle

- En general, nos interesará usar "**while**" cuando puede que la parte repetitiva no se llegue a repetir nunca (por ejemplo: cuando leemos un fichero, si el fichero está vacío, no habrá datos que leer).
- De igual modo, "**do...while**" será lo adecuado cuando debamos repetir al menos una vez (por ejemplo, para pedir una clave de acceso, se le debe preguntar al menos una vez al usuario, o quizás más veces, si la teclea correctamente).
- En cuanto a "**for**", es equivalente a un "while", pero la sintaxis habitual de la orden "for" hace que sea especialmente útil cuando sabemos exactamente cuantas veces queremos que se repita (por ejemplo: 10 veces sería "for (i=1; i<=10; i++)").

Ejercicios propuestos:

- Crear un programa que dé al usuario la oportunidad de adivinar un número del 1 al 100 (prefijado en el programa) en un máximo de 6 intentos. En cada pasada deberá avisar de si se ha pasado o se ha quedado corto.
- Crear un programa que descomponga un número (que teclee el usuario) como producto de sus factores primos. Por ejemplo, $60 = 2 \cdot 2 \cdot 3 \cdot 5$

3. Tipos de datos básicos

3.1. Tipo de datos entero y carácter

Hemos hablado de números enteros, de cómo realizar operaciones sencillas y de cómo usar variables para reservar espacio y poder trabajar con datos cuyo valor no sabemos de antemano.

Empieza a ser el momento de refinar, de dar más detalles. El primer "matiz" importante que nos hemos saltado es el tamaño de los números que podemos emplear, así como su signo (positivo o negativo). Por ejemplo, un dato de tipo "int" puede guardar números de hasta unas nueve cifras, tanto positivos como negativos, y ocupa 4 bytes en memoria.

(**Nota:** si no sabes lo que es un byte, deberías mirar el Apéndice 1 de este texto).

Pero no es la única opción. Por ejemplo, si queremos guardar la edad de una persona, no necesitamos usar números negativos, y nos bastaría con 3 cifras, así que es de suponer que existirá algún tipo de datos más adecuado, que desperdicie menos memoria. También existe el caso contrario: un banco puede necesitar manejar números con más de 9 cifras, así que un dato "int" se les quedaría corto. Siendo estrictos, si hablamos de valores monetarios, necesitaríamos usar decimales, pero eso lo dejamos para el siguiente apartado.

3.1.1. Tipos de datos para números enteros

Los tipos de datos enteros que podemos usar en C#, junto con el espacio que ocupan en memoria y el rango de valores que os permiten almacenar son:

Nombre Del Tipo	Tamaño (bytes)	Rango de valores
sbyte	1	-128 a 127
byte	1	0 a 255
short	2	-32768 a 32767
ushort	2	0 a 65535
int	4	-2147483648 a 2147483647
uint	4	0 a 4294967295
long	8	-9223372036854775808 a 9223372036854775807
ulong	8	0 a 18446744073709551615

Como se puede observar en esta tabla, el tipo de dato más razonable para guardar edades sería "byte", que permite valores entre 0 y 255, y ocupa 3 bytes menos que un "int".

3.1.2. Conversiones de cadena a entero

Si queremos obtener estos datos a partir de una cadena de texto, no siempre nos servirá Convert.ToInt32, porque no todos los datos son enteros de 32 bits (4 bytes). Para datos de tipo

"byte" usaríamos Convert.ToByte (sin signo) y ToSByte (con signo), para datos de 2 bytes tenemosToInt16 (con signo) y ToUInt16 (sin signo), y para los de 8 bytes existenToInt64 (con signo) y ToUInt64 (sin signo).

Ejercicios propuestos:

- Preguntar al usuario su edad, que se guardará en un "byte". A continuación, se deberá le decir que no aparece tantos años (por ejemplo, "No apareces 20 años").
- Pedir al usuario dos números de dos cifras ("byte"), calcular su multiplicación, que se deberá guardar en un "ushort", y mostrar el resultado en pantalla.
- Pedir al usuario dos números enteros largos ("long") y mostrar cuánto es su suma, su resta y su producto.

3.1.3. Incremento y decremento

Conocemos la forma de realizar las operaciones aritméticas más habituales. Peor también existe una operación que es muy frecuente cuando se crean programas, y que no tiene un símbolo específico para representarla en matemáticas: incrementar el valor de una variable en una unidad:

```
a = a + 1;
```

Pues bien, en C#, existe una notación más compacta para esta operación, y para la opuesta (el decremento):

a++;	es lo mismo que	a = a+1;
a--;	es lo mismo que	a = a-1;

Pero esto tiene más misterio todavía del que puede parecer en un primer vistazo: podemos distinguir entre "preincremento" y "postincremento". En C# es posible hacer asignaciones como

```
b = a++;
```

Así, si "a" valía 2, lo que esta instrucción hace es dar a "b" el valor de "a" y aumentar el valor de "a". Por tanto, al final tenemos que b=2 y a=3 (**postincremento**: se incrementa "a" tras asignar su valor).

En cambio, si escribimos

```
b = ++a;
```

y "a" valía 2, primero aumentamos "a" y luego los asignamos a "b" (**preincremento**), de modo que a=3 y b=3.

Por supuesto, también podemos distinguir **postdecremento** (a--) y **predecremento** (--a).

Ejercicios propuestos:

- Crear un programa que use tres variables x,y,z. Sus valores iniciales serán 15, -10, 2.147.483.647. Se deberá incrementar el valor de estas variables. ¿Qué valores esperas que se obtengan? Contrástalo con el resultado obtenido por el programa.

- ¿Cuál sería el resultado de las siguientes operaciones? $a=5; b=++a; c=a++; b=b*5; a=a*2;$

Y ya que estamos hablando de las asignaciones, hay que comentar que en C# es posible hacer **asignaciones múltiples**:

```
a = b = c = 1;
```

3.1.4. Operaciones abreviadas: +=

Pero aún hay más. Tenemos incluso formas reducidas de escribir cosas como "a = a+5". Allá van

$a += b;$	es lo mismo que	$a = a+b;$
$a -= b;$	es lo mismo que	$a = a-b;$
$a *= b;$	es lo mismo que	$a = a*b;$
$a /= b;$	es lo mismo que	$a = a/b;$
$a %= b;$	es lo mismo que	$a = a \% b;$

Ejercicios propuestos:

- Crear un programa que use tres variables x,y,z. Sus valores iniciales serán 15, -10, 214. Se deberá incrementar el valor de estas variables en 12, usando el formato abreviado. ¿Qué valores esperas que se obtengan? Contrástalo con el resultado obtenido por el programa.
- ¿Cuál sería el resultado de las siguientes operaciones? $a=5; b=a+2; b-=3; c=-3; c*=2; ++c; a*=b;$

3.2. Tipo de datos real

Cuando queremos almacenar datos con decimales, no nos sirve el tipo de datos "int". Necesitamos otro tipo de datos que sí esté preparado para guardar números "reales" (con decimales). En el mundo de la informática hay dos formas de trabajar con números reales:

- **Coma fija:** el número máximo de cifras decimales está fijado de antemano, y el número de cifras enteras también. Por ejemplo, con un formato de 3 cifras enteras y 4 cifras decimales, el número 3,75 se almacenaría correctamente, el número 970,4361 también, pero el 5,678642 se guardaría como 5,6786 (se perdería a partir de la cuarta cifra decimal) y el 1010 no se podría guardar (tiene más de 3 cifras enteras).
- **Coma flotante:** el número de decimales y de cifras enteras permitido es variable, lo que importa es el número de cifras significativas (a partir del último 0). Por ejemplo, con 5 cifras significativas se podrían almacenar números como el 13405000000 o como el 0,0000007349 pero no se guardaría correctamente el 12,0000034, que se redondearía a un número cercano.

3.2.1. Simple y doble precisión

Tenemos tres tamaños para elegir, según si queremos guardar números con mayor cantidad de cifras o con menos. Para números con pocas cifras significativas (un máximo de 7, lo que se conoce como "un dato real de simple precisión") existe el tipo "float" y para números que necesiten más precisión (unas 15 cifras, "doble precisión") tenemos el tipo "double". En C# existe un tercer tipo de números reales, con mayor precisión todavía, que es el tipo "decimal":

	float	double	decimal
Tamaño en bits	32	64	128
Valor más pequeño	$-1,5 \cdot 10^{-45}$	$5,0 \cdot 10^{-324}$	$1,0 \cdot 10^{-28}$
Valor más grande	$3,4 \cdot 10^{38}$	$1,7 \cdot 10^{308}$	$7,9 \cdot 10^{28}$
Cifras significativas	7	15-16	28-29

Para definirlos, se hace igual que en el caso de los números enteros:

```
float x;
```

o bien, si queremos dar un valor inicial en el momento de definirlos (recordando que para las cifras decimales no debemos usar una coma, sino un punto):

```
float x = 12.56;
```

3.2.2. Mostrar en pantalla números reales

Al igual que hacíamos con los enteros, podemos leer como cadena de texto, y convertir cuando vayamos a realizar operaciones aritméticas. Ahora usaremos Convert.ToDouble cuando se trate de un dato de doble precisión, Convert.ToSingle cuando sea un dato de simple precisión (float) y Convert.ToDecimal para un dato de precisión extra (decimal):

```
/*
 *----- Ejemplo en C# nº 27:
 *----- ejemplo27.cs
 *----- Números reales (1)
 *----- Introducción a C#,
 *----- Nacho Cabanes
 *----- */

using System;

public class Ejemplo27
{
    public static void Main()
    {
        float primerNumero;
        float segundoNumero;
        float suma;

        Console.WriteLine("Introduce el primer número");
        primerNumero = Convert.ToSingle(Console.ReadLine());
        Console.WriteLine("Introduce el segundo número");
```

```

        segundoNumero = Convert.ToSingle(Console.ReadLine());
        suma = primerNumero + segundoNumero;

        Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}

```

Cuidado al probar este programa: aunque en el fuente debemos escribir los decimales usando un punto, como 123.456, al poner el ejecutable en marcha parte del trabajo se le encarga al sistema operativo, de modo que si éste sabe que en nuestro país se usa la coma para los decimales, considere la coma el separador correcto y no el punto, como ocurre si introducimos estos datos en la versión española de Windows XP:

```

ejemplo05
Introduce el primer número
23,6
Introduce el segundo número
34.2
La suma de 23,6 y 342 es 365,6

```

3.2.3. Formatear números

En más de una ocasión nos interesará formatear los números para mostrar una cierta cantidad de decimales: por ejemplo, nos puede interesar que una cifra que corresponde a dinero se muestre siempre con dos cifras decimales, o que una nota se muestre redondeada, sin decimales.

Una forma de conseguirlo es crear una cadena de texto a partir del número, usando "ToString". A esta orden se le puede indicar un dato adicional, que es el formato numérico que queremos usar, por ejemplo: suma.ToString("0.00")

Algunas de los códigos de formato que se pueden usar son:

- Un cero (0) indica una posición en la que debe aparecer un número, y se mostrará un 0 si no hay ninguno.
- Una almohadilla (#) indica una posición en la que puede aparecer un número, y no se escribirá nada si no hay número.
- Un punto (.) indica la posición en la que deberá aparecer la coma decimal.
- Alternativamente, se pueden usar otros formatos abreviados: por ejemplo, N2 quiere decir "con dos cifras decimales" y N5 es "con cinco cifras decimales"

Vamos a probarlos en un ejemplo:

```

/*
 *-----*/
/* Ejemplo en C# nº 28:      */
/* ejemplo28.cs               */
/*                           */
/* Formato de núms. reales   */
/*                           */
/* Introducción a C#,          */
*/

```

```

/*
 * Nacho Cabanes
 */
using System;

public class Ejemplo28
{
    public static void Main()
    {
        double numero = 12.34;

        Console.WriteLine( numero.ToString("N1") );
        Console.WriteLine( numero.ToString("N3") );
        Console.WriteLine( numero.ToString("0.0") );
        Console.WriteLine( numero.ToString("0.000") );
        Console.WriteLine( numero.ToString("#.#") );
        Console.WriteLine( numero.ToString("#.###") );
    }
}

```

El resultado de este ejemplo sería:

```

12,3
12,340
12,3
12,340
12,3
12,34

```

Como se puede ver, ocurre lo siguiente:

- Si indicamos menos decimales de los que tiene el número, se redondea.
- Si indicamos más decimales de los que tiene el número, se mostrarán ceros si usamos como formato Nx o 0.000, y no se mostrará nada si usamos #.###
- Si indicamos menos cifras antes de la coma decimal de las que realmente tiene el número, aun así se muestran todas ellas.

Ejercicios propuestos:

- El usuario de nuestro programa podrá teclear dos números de hasta 12 cifras significativas. El programa deberá mostrar el resultado de dividir el primer número entre el segundo, utilizando tres cifras decimales.
- Crear un programa que use tres variables x,y,z. Las tres serán números reales, y nos bastará con dos cifras decimales. Deberá pedir al usuario los valores para las tres variables y mostrar en pantalla el valor de $x^2 + y - z$ (con exactamente dos cifras decimales).

Un uso alternativo de `ToString` es el de **cambiar un número de base**. Por ejemplo, habitualmente trabajamos con números decimales (en base 10), pero en informática son también muy frecuentes la base 2 (el sistema binario) y la base 16 (el sistema hexadecimal). Podemos

convertir un número a binario o hexadecimal (o a base octal, menos frecuente) usando `Convert.ToString` e indicando la base, como en este ejemplo:

```
/*
 * Ejemplo en C# nº 28b:
 */
/* ejemplo28b.cs */
/*
 */
/* Hexadecimal y binario */
/*
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes */
/*
 */

using System;

public class Ejemplo28b
{
    public static void Main()
    {
        int numero = 247;

        Console.WriteLine( Convert.ToString(numero, 16) );
        Console.WriteLine( Convert.ToString(numero, 2) );
    }
}
```

Su resultado sería:

```
f7
11110111
```

(Si quieres saber más sobre el sistema hexadecimal, mira los apéndices al final de este texto)

3.3. Tipo de datos carácter

También tenemos un tipo de datos que nos permite almacenar una única letra, el tipo "char":

```
char letra;
```

Asignar valores es sencillo: el valor se indica entre comillas simples

```
letra = 'a';
```

Para leer valores desde teclado, lo podemos hacer de forma similar a los casos anteriores: leemos toda una frase con `ReadLine` y convertimos a tipo "char" usando `Convert.ToChar`:

```
letra = Convert.ToChar(Console.ReadLine());
```

Así, un programa que de un valor inicial a una letra, la muestre, lea una nueva letra tecleada por el usuario, y la muestre, podría ser:

```

/*
/* Ejemplo en C# nº 29: */
/* ejemplo29.cs */
/*
*/
/* Tipo de datos "char" */
/*
*/
/* Introducción a C#, */
/* Nacho Cabanes */
/*
*/
using System;

public class Ejemplo29
{
    public static void Main()
    {
        char letra;

        letra = 'a';
        Console.WriteLine("La letra es {0}", letra);

        Console.WriteLine("Introduce una nueva letra");
        letra = Convert.ToChar(Console.ReadLine());
        Console.WriteLine("Ahora la letra es {0}", letra);
    }
}

```

3.3.1. Secuencias de escape: \n y otras.

Como hemos visto, los textos que aparecen en pantalla se escriben con WriteLine, indicados entre paréntesis y entre comillas dobles. Entonces surge una dificultad: ¿cómo escribimos una comilla doble en pantalla? La forma de conseguirlo es usando ciertos caracteres especiales, lo que se conoce como "secuencias de escape". Existen ciertos caracteres especiales que se pueden escribir después de una barra invertida (\) y que nos permiten conseguir escribir esas comillas dobles y algún otro carácter poco habitual. Por ejemplo, con \" se escribirán unas comillas dobles, y con \' unas comillas simples, o con \n se avanzará a la línea siguiente de pantalla.

Estas secuencias especiales son las siguientes:

Secuencia	Significado
\a	Emite un pitido
\b	Retroceso (permite borrar el último carácter)
\f	Avance de página (expulsa una hoja en la impresora)
\n	Avanza de línea (salta a la línea siguiente)
\r	Retorno de carro (va al principio de la línea)
\t	Salto de tabulación horizontal
\v	Salto de tabulación vertical
'	Muestra una comilla simple
"	Muestra una comilla doble
\\"	Muestra una barra invertida
\0	Carácter nulo (NULL)

Vamos a ver un ejemplo que use los más habituales:

```
/*
 * Ejemplo en C# nº 30:
 */
/* ejemplo30.cs */
/*
 */
/* Secuencias de escape */
/*
 */
/* Introduccion a C#, */
/* Nacho Cabanes */
/*
 */

using System;

public class Ejemplo30
{
    public static void Main()
    {
        Console.WriteLine("Esta es una frase");
        Console.WriteLine();
        Console.WriteLine();
        Console.WriteLine("y esta es otra, separada dos lineas");

        Console.WriteLine("\n\nJuguemos mas:\n\notro salto");
        Console.WriteLine("Comillas dobles: \" y simples ', y barra \\\"");
    }
}
```

Su resultado sería este:

Esta es una frase

y esta es otra, separada dos lineas

Juguemos mas:

otro salto

Comillas dobles: " y simples ', y barra \

En algunas ocasiones puede ser incómodo manipular estas secuencias de escape. Por ejemplo, cuando usemos estructuras de directorios: c:\\datos\\ejemplos\\curso\\ejemplo1. En este caso, se puede usar una arroba (@) antes del texto sin las barras invertidas:

```
ruta = @"c:\\datos\\ejemplos\\curso\\ejemplo1"
```

En este caso, el problema está si aparecen comillas en medio de la cadena. Para solucionarlo, se duplican las comillas, así:

```
orden = "copy ""documento de ejemplo"" f:"
```

Ejercicio propuesto: Crear un programa que pida al usuario que teclee cuatro letras y las muestre en pantalla juntas, pero en orden inverso, y entre comillas dobles. Por ejemplo si las letras que se teclean son a, l, o, h, escribiría "hola".

3.4. Toma de contacto con las cadenas de texto

Las cadenas de texto son tan fáciles de manejar como los demás tipos de datos que hemos visto, con apenas tres diferencias:

- Se declaran con "string".
- Si queremos dar un valor inicial, éste se indica entre comillas dobles.
- Cuando leemos con ReadLine, no hace falta convertir el valor obtenido.

Así, un ejemplo que diera un valor a un "string", lo mostrara (entre comillas, para practicar las secuencias de escape que hemos visto en el apartado anterior) y leyera un valor tecleado por el usuario podría ser:

```
/*-----*/
/* Ejemplo en C# nº 31:      */
/* ejemplo31.cs              */
/*                         */
/* Uso basico de "string"   */
/*                         */
/* Introduccion a C#,        */
/* Nacho Cabanes            */
/*-----*/
using System;

public class Ejemplo31
{
    public static void Main()
    {
        string frase;

        frase = "Hola, como estas?";
        Console.WriteLine("La frase es \"{0}\", frase);
```

```

        Console.WriteLine("Introduce una nueva frase");
        frase = Console.ReadLine();
        Console.WriteLine("Ahora la frase es \"{0}\", frase");
    }
}

```

Se pueden hacer muchas más operaciones sobre cadenas de texto: convertir a mayúsculas o a minúsculas, eliminar espacios, cambiar una subcadena por otra, dividir en trozos, etc. Pero ya volveremos a las cadenas más adelante, cuando conozcamos las estructuras de control básicas.

3.5. Los valores "booleanos"

En C# tenemos también un tipo de datos llamado "booleano" ("bool"), que puede tomar dos valores: verdadero ("true") o falso ("false"):

```

bool encontrado;
encontrado = true;

```

Este tipo de datos hará que podamos escribir de forma sencilla algunas condiciones que podrían resultar complejas. Así podemos hacer que ciertos fragmentos de nuestro programa no sean "if ((vidas==0) || (tiempo == 0) || ((enemigos ==0) && (nivel == ultimoNivel)))" sino simplemente "if (partidaTerminada) ..."

A las variables "bool" también se le puede dar como valor el resultado de una comparación:

```

partidaTerminada = false;
partidaTerminada = (enemigos ==0) && (nivel == ultimoNivel);
if (vidas == 0) partidaTerminada = true;

```

Lo emplearemos a partir de ahora en los fuentes que usen condiciones un poco complejas. Un ejemplo que pida una letra y diga si es una vocal, una cifra numérica u otro símbolo, usando variables "bool" podría ser:

```

/*
 *----- Ejemplo en C# nº 32:
 *----- ejemplo32.cs
 */
/*
 */
/* Condiciones con if (8)
 */
/* Variables bool
 */
/*
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes
 */
*/

```

```

using System;

public class Ejemplo32
{
    public static void Main()
    {

```

```
char letra;
bool esVocal, esCifra;

Console.WriteLine("Introduce una letra");
letra = Convert.ToChar(Console.ReadLine());

esCifra = (letra >= '0') && (letra <='9');

esVocal = (letra == 'a') || (letra == 'e') || (letra == 'i') ||
          (letra == 'o') || (letra == 'u');

if (esCifra)
    Console.WriteLine("Es una cifra numérica.");
else if (esVocal)
    Console.WriteLine("Es una vocal.");
else
    Console.WriteLine("Es una consonante u otro número.");
}
```

Ejercicios propuestos:

- Crear un programa que use el operador condicional para dar a una variable llamada "iguales" (booleana) el valor "true" si los dos números que ha tecleado el usuario son iguales, o "false" si son distintos.

4. Arrays, estructuras y cadenas de texto

4.1. Conceptos básicos sobre arrays o tablas

4.1.1. Definición de un array y acceso a los datos

Una tabla, vector, matriz o **array** (que algunos autores traducen por "**arreglo**") es un conjunto de elementos, todos los cuales son del mismo tipo. Estos elementos tendrán todos el mismo nombre, y ocuparán un espacio contiguo en la memoria.

Por ejemplo, si queremos definir un grupo de números enteros, el tipo de datos que usaremos para declararlo será "int[]". Si sabemos desde el principio cuantos datos tenemos (por ejemplo 4), les reservaremos espacio con "= new int[4]", así

```
int[] ejemplo = new int[4];
```

Podemos acceder a cada uno de los valores individuales indicando su nombre (ejemplo) y el número de elemento que nos interesa, pero con una precaución: se empieza a numerar desde 0, así que en el caso anterior tendríamos 4 elementos, que serían ejemplo[0], ejemplo[1], ejemplo[2], ejemplo[3].

Como ejemplo, vamos a definir un grupo de 5 números enteros y hallar su suma:

```
/*-----*/
/* Ejemplo en C# nº 33:      */
/* ejemplo33.cs               */
/*                           */
/* Primer ejemplo de tablas */
/*                           */
/* Introduccion a C#,        */
/* Nacho Cabanes             */
/*-----*/
using System;

public class Ejemplo33
{
    public static void Main()
    {
        int[] numero = new int[5];          /* Un array de 5 números enteros */
        int suma;                         /* Un entero que será la suma */

        numero[0] = 200;                  /* Les damos valores */
        numero[1] = 150;
        numero[2] = 100;
        numero[3] = -50;
        numero[4] = 300;
        suma = numero[0] + /* Y hallamos la suma */
               numero[1] + numero[2] + numero[3] + numero[4];
        Console.WriteLine("Su suma es {0}", suma);
        /* Nota: esta es la forma más ineficiente e incómoda */
        /* Ya lo iremos mejorando */
    }
}
```

```
 }
```

Ejercicios propuestos:

- Un programa que pida al usuario 4 números, los memorice (utilizando una tabla), calcule su media aritmética y después muestre en pantalla la media y los datos tecleados.
- Un programa que pida al usuario 5 números reales y luego los muestre en el orden contrario al que se introdujeron.
- Un programa que pida al usuario 10 números enteros y calcule (y muestre) cuál es el mayor de ellos.

4.1.2. Valor inicial de un array

Al igual que ocurría con las variables "normales", podemos dar valor a los elementos de una tabla al principio del programa. Será más cómodo que dar los valores uno por uno, como hemos hecho antes, pero sólo se podrá hacer si conocemos todos los valores. En este caso, los indicaremos todos entre llaves, separados por comas:

```
/*
 * Ejemplo en C# nº 34:
 */
/* ejemplo34.cs
 */
/* Segundo ejemplo de
 */
/* tablas
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes
 */
/*-----*/
using System;

public class Ejemplo34
{
    public static void Main()
    {
        int[] numero =           /* Un array de 5 números enteros */
        {200, 150, 100, -50, 300};
        int suma;                /* Un entero que será la suma */

        suma = numero[0] +      /* Y hallamos la suma */
              numero[1] + numero[2] + numero[3] + numero[4];
        Console.WriteLine("Su suma es {0}", suma);
        /* Nota: esta forma es algo menos engorrosa, pero todavía no */
        /* está bien hecho. Lo seguiremos mejorando */
    }
}
```

Ejercicios propuestos:

- Un programa que almacene en una tabla el número de días que tiene cada mes (supondremos que es un año no bisiesto), pida al usuario que le indique un mes (1=enero, 12=diciembre) y muestre en pantalla el número de días que tiene ese mes.
- Un programa que almacene en una tabla el número de días que tiene cada mes (año no bisiesto), pida al usuario que le indique un mes (ej. 2 para febrero) y un día (ej. el día 15) y diga qué número de día es dentro del año (por ejemplo, el 15 de febrero sería el día número 46, el 31 de diciembre sería el día 365).

4.1.3. Recorriendo los elementos de una tabla

Es de esperar que exista una forma más cómoda de acceder a varios elementos de un array, sin tener siempre que repetirlos todos, como hemos hecho en

```
suma = numero[0] + numero[1] + numero[2] + numero[3] + numero[4];
```

El "truco" consistirá en emplear cualquiera de las estructuras repetitivas que ya hemos visto (while, do..while, for), por ejemplo así:

```
suma = 0; /* Valor inicial de la suma */
for (i=0; i<=4; i++) /* Y hallamos la suma repetitiva */
    suma += numero[i];
```

El fuente completo podría ser así:

```
/*-----*/
/* Ejemplo en C# nº 35: */
/* ejemplo35.cs */
/*
/* Tercer ejemplo de */
/* tablas */
/*
/* Introducción a C#,
/* Nacho Cabanes
/*-----*/

using System;

public class Ejemplo35
{
    public static void Main()
    {

        int[] numero = /* Un array de 5 números enteros */
            {200, 150, 100, -50, 300};
        int suma; /* Un entero que será la suma */
        int i; /* Para recorrer los elementos */

        suma = 0; /* Valor inicial de la suma */
        for (i=0; i<=4; i++) /* Y hallamos la suma repetitiva */
            suma += numero[i];

        Console.WriteLine("Su suma es {0}", suma);
    }
}
```

```

}
```

En este caso, que sólo sumábamos 5 números, no hemos escrito mucho menos, pero si trabajásemos con 100, 500 o 1000 números, la ganancia en comodidad sí que está clara.

4.1.4. Datos repetitivos introducidos por el usuario

Si queremos que sea el usuario el que introduzca datos a un array, usaríamos otra estructura repetitiva ("for", por ejemplo) para pedírselos:

```

/*
 * Ejemplo en C# nº 36:
 * ejemplo36.cs
 *
 * Cuarto ejemplo de
 * tablas
 *
 * Introduccion a C#,
 * Nacho Cabanes
 */

using System;

public class Ejemplo36
{
    public static void Main()
    {

        int[] numero = new int[5]; /* Un array de 5 números enteros */
        int suma; /* Un entero que será la suma */
        int i; /* Para recorrer los elementos */

        for (i=0; i<=4; i++) /* Pedimos los datos */
        {
            Console.WriteLine("Introduce el dato numero {0}: ", i+1);
            numero[i] = Convert.ToInt32(Console.ReadLine());
        }

        suma = 0; /* Valor inicial de la suma */
        for (i=0; i<=4; i++) /* Y hallamos la suma repetitiva */
            suma += numero[i];

        Console.WriteLine("Su suma es {0}", suma);
    }
}

```

Ejercicios propuestos:

- A partir del programa propuesto en 4.1.2, que almacenaba en una tabla el número de días que tiene cada mes, crear otro que pida al usuario que le indique la fecha, detallando el día (1 al 31) y el mes (1=enero, 12=diciembre), como respuesta muestre en pantalla el número de días que quedan hasta final de año.

- Crear un programa que pida al usuario 10 números y luego los muestre en orden inverso (del último que se ha introducido al primero que se introdujo).
- Un programa que pida al usuario 10 números y luego calcule y muestre cual es el mayor de todos ellos.
- Un programa que pida al usuario 10 números, calcule su media y luego muestre los que están por encima de la media.
- Un programa que pida 10 nombres y los memorice (pista: esta vez se trata de un array de "string"). Despues deberá pedir que se teclee un nombre y dirá si se encuentra o no entre los 10 que se han tecleado antes. Volverá a pedir otro nombre y a decir si se encuentra entre ellos, y así sucesivamente hasta que se teclee "fin".
- Un programa que prepare espacio para un máximo de 100 nombres. El usuario deberá ir introduciendo un nombre cada vez, hasta que se pulse Intro sin teclear nada, momento en el que dejarán de pedirse más nombres y se mostrará en pantalla la lista de los nombres que se han introducido.
- Un programa que准备 espacio para un máximo de 10 nombres. Deberá mostrar al usuario un menú que le permita realizar las siguientes operaciones:
 - Añadir un dato al final de los ya existentes.
 - Insertar un dato en una cierta posición (lo que quedén detrás deberán desplazarse "a la derecha" para dejarle hueco; por ejemplo, si el array contiene "hola", "adios" y se pide insertar "bien" en la segunda posición, el array pasará a contener "hola", "bien", "adios".
 - Borrar el dato que hay en una cierta posición (lo que estaban detrás deberán desplazarse "a la izquierda" para que no haya huecos; por ejemplo, si el array contiene "hola", "bien", "adios" y se pide borrar el dato de la segunda posición, el array pasará a contener "hola", "adios"
 - Mostrar los datos que contiene el array.
 - Salir del programa.

4.2. Tablas bidimensionales

Podemos declarar tablas de **dos o más dimensiones**. Por ejemplo, si queremos guardar datos de dos grupos de alumnos, cada uno de los cuales tiene 20 alumnos, tenemos dos opciones:

- Podemos usar `int datosAlumnos[40]` y entonces debemos recordar que los 20 primeros datos corresponden realmente a un grupo de alumnos y los 20 siguientes a otro grupo. Es "demasiado artesanal", así que no daremos más detalles.
- O bien podemos emplear `int datosAlumnos[2,20]` y entonces sabemos que los datos de la forma `datosAlumnos[0,i]` son los del primer grupo, y los `datosAlumnos[1,i]` son los del segundo.
- Una alternativa, que puede sonar más familiar a quien ya haya programado en C es emplear `int datosAlumnos[2][20]` pero en C# esto no tiene exactamente el mismo significado que [2,20], sino que se trata de dos arrays, cuyos elementos a su vez son arrays de 20 elementos. De hecho, podrían ser incluso dos arrays de distinto tamaño, como veremos en el segundo ejemplo.

En cualquier caso, si queremos indicar valores iniciales, lo haremos entre llaves, igual que si fuera una tabla de una única dimensión.

Vamos a ver un primer ejemplo del uso con arrays de la forma [2,20], lo que podríamos llamar el "estilo Pascal", en el usemos tanto arrays con valores prefijados, como arrays para los que reservemos espacio con "new" y a los que demos valores más tarde:

```
/*-----*/
/* Ejemplo en C# nº 37:      */
/* ejemplo37.cs               */
/*                           */
/* Array de dos dimensiones */
/* al estilo Pascal           */
/*                           */
/* Introducción a C#,          */
/* Nacho Cabanes              */
/*-----*/
using System;

public class Ejemplo37
{
    public static void Main()
    {

        int[,] notas1 = new int[2,2]; // 2 bloques de 2 datos
        notas1[0,0] = 1;
        notas1[0,1] = 2;
        notas1[1,0] = 3;
        notas1[1,1] = 4;

        int[,] notas2 = // 2 bloques de 10 datos
        {
            {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
            {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
        };

        Console.WriteLine("La nota1 del segundo alumno del grupo 1 es {0}",
            notas1[0,1]);
        Console.WriteLine("La nota2 del tercer alumno del grupo 1 es {0}",
            notas2[0,2]);
    }
}
```

Este tipo de tablas de varias dimensiones son las que se usan también para guardar matrices, cuando se trata de resolver problemas matemáticos más complejos que los que hemos visto hasta ahora.

La otra forma de tener arrays multidimensionales son los "arrays de arrays", que, como ya hemos comentado, y como veremos en este ejemplo, pueden tener elementos de distinto tamaño. En ese caso nos puede interesar saber su longitud, para lo que podemos usar "a.Length":

```
/*-----*/
```

```

/*
 * Ejemplo en C# nº 38:
 */
/*
 * Array de dos dimensiones
 */
/*
 * al estilo C... o casi
 */
/*
 * Introducción a C#,
 */
/*
 * Nacho Cabanes
 */
/*-----*/
using System;

public class Ejemplo38
{
    public static void Main()
    {

        int[][] notas;          // Array de dos dimensiones
        notas = new int[3][];   // Serán 3 bloques de datos
        notas[0] = new int[10]; // 10 notas en un grupo
        notas[1] = new int[15]; // 15 notas en otro grupo
        notas[2] = new int[12]; // 12 notas en el último

        // Damos valores de ejemplo
        for (int i=0;i<notas.Length;i++)
        {
            for (int j=0;j<notas[i].Length;j++)
            {
                notas[i][j] = i + j;
            }
        }

        // Y mostramos esos valores
        for (int i=0;i<notas.Length;i++)
        {
            for (int j=0;j<notas[i].Length;j++)
            {
                Console.Write(" {0}", notas[i][j]);
            }
            Console.WriteLine();
        }
    }
}

```

La salida de este programa sería

```

0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 3 4 5 6 7 8 9 10 11 12 13

```

Ejercicios propuestos:

- Un programa que al usuario dos bloques de 10 números enteros (usando un array de dos dimensiones). Después deberá mostrar el mayor dato que se ha introducido en cada uno de ellos.

- Un programa que al usuario dos bloques de 6 cadenas de texto. Después pedirá al usuario una nueva cadena y comprobará si aparece en alguno de los dos bloques de información anteriores.

4.3. Estructuras o registros

4.3.1. Definición y acceso a los datos

Un **registro** es una agrupación de datos, los cuales no necesariamente son del mismo tipo. Se definen con la palabra "**struct**". La serie de datos que van a formar

En C# (al contrario que en C), primero deberemos declarar cual va a ser la estructura de nuestro registro, lo que no se puede hacer dentro de "Main". Más adelante, ya dentro de "Main", podremos declarar variables de ese nuevo tipo.

Los datos que forman un "struct" pueden ser públicos o privados. Por ahora, a nosotros nos interesará que sean accesibles desde el resto de nuestro programa, por lo que les añadiremos delante la palabra "public" para indicar que queremos que sean públicos.

Ya desde el cuerpo del programa, para acceder a cada uno de los datos que forman el registro, tanto si queremos leer su valor como si queremos cambiarlo, se debe indicar el nombre de la variable y el del dato (o campo) separados por un punto:

```
/*
 *----- Ejemplo en C# nº 39: -----
 */
/* ejemplo39.cs */
/*
 */
/* Registros (struct) */
/*
 */
/* Introduccion a C#, */
/* Nacho Cabanes */
/*----- */

using System;

public class Ejemplo39
{

    struct tipoPersona
    {
        public string nombre;
        public char inicial;
        public int edad;
        public float nota;
    }

    public static void Main()
    {
        tipoPersona persona;

        persona.nombre = "Juan";
        persona.inicial = 'J';
        persona.edad = 20;
        persona.nota = 7.5f;
    }
}
```

```

        Console.WriteLine("La edad de {0} es {1}",
            persona.nombre, persona.edad);
    }
}

```

Nota: La notación 7.5f se usa para detallar que se trata de un número real de simple precisión (un "float"), porque de lo contrario, 7.5 se consideraría un número de doble precisión, y al tratar de compilar obtendríamos un mensaje de error, diciendo que no se puede convertir de "double" a "float" sin pérdida de precisión. Al añadir la "f" al final, estamos diciendo "quiero que éste número se tome como un float; sé que habrá una pérdida de precisión pero es aceptable para mí".

Ejercicios propuestos:

- Un "struct" que almacene datos de una canción en formato MP3: Artista, Título, Duración (en segundos), Tamaño del fichero (en KB). Un programa debe pedir los datos de una canción al usuario, almacenarlos en dicho "struct" y después mostrarlos en pantalla.

4.3.2. Arrays de estructuras

Hemos guardado varios datos de una persona. Se pueden almacenar los de **varias personas** si combinamos el uso de los "struct" con las tablas (arrays) que vimos anteriormente. Por ejemplo, si queremos guardar los datos de 100 personas podríamos hacer:

```

/*
 *----- Ejemplo en C# nº 40: -----
 */
/* ejemplo40.cs */
/*
 */
/* Array de struct */
/*
 */
/* Introduccion a C#, */
/* Nacho Cabanes */
/*-----*/
using System;

public class Ejemplo40
{
    struct tipoPersona
    {
        public string nombre;
        public char inicial;
        public int edad;
        public float nota;
    }

    public static void Main()
    {
        tipoPersona[] persona = new tipoPersona[100];

        persona[0].nombre = "Juan";
        persona[0].inicial = 'J';
        persona[0].edad = 20;
        persona[0].nota = 7.5f;
    }
}

```

```

Console.WriteLine("La edad de {0} es {1}",
    persona[0].nombre, persona[0].edad);

persona[1].nombre = "Pedro";
Console.WriteLine("La edad de {0} es {1}",
    persona[1].nombre, persona[1].edad);
}
}

```

La inicial de la primera persona sería "persona[0].inicial", y la edad del último sería "persona[99].edad".

Al probar este programa obtenemos

```

La edad de Juan es 20
La edad de Pedro es 0

```

Porque cuando reservamos espacio para los elementos de un "array" usando "new", sus valores se dejan "vacíos" (0 para los números, cadenas vacías para las cadenas de texto).

Ejercicios propuestos:

- Ampliar el programa del apartado 4.3.1, para que almacene datos de hasta 100 canciones. Deberá tener un menú que permita las opciones: añadir una nueva canción, mostrar el título de todas las canciones, buscar la canción que contenga un cierto texto (en el artista o en el título).
- Un programa que permita guardar datos de "imágenes" (ficheros de ordenador que contengan fotografías o cualquier otro tipo de información gráfica). De cada imagen se debe guardar: nombre (texto), ancho en píxeles (por ejemplo 2000), alto en píxeles (por ejemplo, 3000), tamaño en Kb (por ejemplo 145,6). El programa debe ser capaz de almacenar hasta 700 imágenes (deberá avisar cuando su capacidad esté llena). Debe permitir las opciones: añadir una ficha nueva, ver todas las fichas (número y nombre de cada imagen), buscar la ficha que tenga un cierto nombre.

4.3.3. Estructuras anidadas

Podemos encontrarnos con un registro que tenga varios datos, y que a su vez ocurra que uno de esos datos esté formado por varios datos más sencillos. Por ejemplo, una fecha de nacimiento podría estar formada por día, mes y año. Para hacerlo desde C#, incluiríamos un "struct" dentro de otro, así:

```

/*-----*/
/* Ejemplo en C# nº 41: */
/* ejemplo41.cs */
/*
/* Registros anidados */
/*
/* Introduccion a C#,
/* Nacho Cabanes
/*-----*/

```

```

using System;

public class Ejemplo41
{
    struct fechaNacimiento
    {
        public int dia;
        public int mes;
        public int anyo;
    }

    struct tipoPersona
    {
        public string nombre;
        public char inicial;
        public fechaNacimiento diaDeNacimiento;
        public float nota;
    }

    public static void Main()
    {
        tipoPersona persona;

        persona.nombre = "Juan";
        persona.inicial = 'J';
        persona.diaDeNacimiento.dia = 15;
        persona.diaDeNacimiento.mes = 9;
        persona.nota = 7.5f;
        Console.WriteLine("{0} nació en el mes {1}",
            persona.nombre, persona.diaDeNacimiento.mes);
    }
}

```

Ejercicios propuestos:

- Ampliar el programa del primer apartado de 4.3.2, para que el campo "duración" se almacene como minutos y segundos, usando un "struct" anidado que contenga a su vez estos dos campos.

4.4. Cadenas de caracteres

4.4.1. Definición. Lectura desde teclado

Hemos visto cómo leer cadenas de caracteres (Console.ReadLine) y cómo mostrarlas en pantalla (Console.Write), así como la forma de darles un valor(=). También podemos comparar cual es su valor, usando ==, o formar una cadena a partir de otras si las unimos con el símbolo de la suma (+):

Así, un ejemplo que nos pidiese nuestro nombre y nos saludase usando todas estas posibilidades podría ser:

```
/*-----*/
```

```

/*
/* Ejemplo en C# nº 42:      */
/* ejemplo42.cs               */
/*
/* Cadenas de texto (1)       */
/*
/* Introduccion a C#,          */
/* Nacho Cabanes               */
/*-----*/
using System;

public class Ejemplo42
{
    public static void Main()
    {
        string saludo = "Hola";
        string segundoSaludo;
        string nombre, despedida;

        segundoSaludo = "Que tal?";
        Console.WriteLine("Dime tu nombre... ");
        nombre = Console.ReadLine();

        Console.WriteLine("{0} {1}", saludo, nombre);
        Console.WriteLine(segundoSaludo);

        if (nombre == "Alberto")
            Console.WriteLine("Dices que eres Alberto?");
        else
            Console.WriteLine("Así que no eres Alberto?");

        despedida = "Adios " + nombre + "!";
        Console.WriteLine(despedida);
    }
}

```

4.4.2. Cómo acceder a las letras que forman una cadena

Podemos leer (o modificar) una de las letras de una cadena de igual forma que leemos los elementos de cualquier array: si la cadena se llama "texto", el primer elemento será texto[0], el segundo será texto[1] y así sucesivamente.

Eso sí, las cadenas en C# no se pueden modificar letra a letra: no podemos hacer texto[0]='a'. Para eso habrá que usar una construcción auxiliar, que veremos más adelante.

4.4.3. Longitud de la cadena.

Podemos saber cuantas letras forman una cadena con "cadena.Length". Esto permite que podamos recorrer la cadena letra por letra, usando construcciones como "for".

Ejercicios propuestos:

- Un programa que te pida tu nombre y lo muestre en pantalla separando cada letra de la siguiente con un espacio. Por ejemplo, si tu nombre es "Juan", debería aparecer en pantalla "J u a n".
- Un programa capaz de sumar dos números enteros muy grandes (por ejemplo, de 30 cifras), que se deberán pedir como cadena de texto y analizar letra a letra.
- Un programa capaz de multiplicar dos números enteros muy grandes (por ejemplo, de 30 cifras), que se deberán pedir como cadena de texto y analizar letra a letra.

4.4.4. Extraer una subcadena

Podemos extraer parte del contenido de una cadena con "Substring", que recibe dos parámetros: la posición a partir de la que queremos empezar y la cantidad de caracteres que queremos obtener. El resultado será otra cadena:

```
saludo = frase.Substring(0, 4);
```

Podemos omitir el segundo número, y entonces se extraerá desde la posición indicada hasta el final de la cadena.

4.4.5. Buscar en una cadena

Para ver si una cadena contiene un cierto texto, podemos usar IndexOf ("posición de"), que nos dice en qué posición se encuentra (o devuelve el valor -1 si no aparece):

```
if (nombre.IndexOf("Juan") >= 0) Console.WriteLine("Bienvenido, Juan");
```

Podemos añadir un segundo parámetro opcional, que es la posición a partir de la que queremos buscar:

```
if (nombre.IndexOf("Juan", 5) >= 0) ...
```

La búsqueda termina al final de la cadena, salvo que indiquemos que termine antes con un tercer parámetro opcional:

```
if (nombre.IndexOf("Juan", 5, 15) >= 0) ...
```

De forma similar, LastIndexOf ("última posición de") indica la última aparición (es decir, busca de derecha a izquierda).

4.4.6. Otras manipulaciones de cadenas

Ya hemos comentado que las cadenas en C# son inmutables, no se pueden modificar. Pero sí podemos realizar ciertas operaciones sobre ellas para obtener una nueva cadena. Por ejemplo:

- ToUpper() convierte a mayúsculas: nombreCorrecto = nombre.ToUpper();
- ToLower() convierte a minúsculas: password2 = password.ToLower();

- Insert(int posición, string subcadena): Insertar una subcadena en una cierta posición de la cadena inicial: nombreFormal = nombre.Insert(0,"Don");
- Remove(int posición, int cantidad): Elimina una cantidad de caracteres en cierta posición: apellidos = nombreCompleto.Remove(0,6);
- Replace(string textoASustituir, string cadenaSustituta): Sustituye una cadena (todas las veces que aparezca) por otra: nombreCorregido = nombre.Replace("Pepe", "Jose");

Un programa que probara todas estas posibilidades podría ser así:

```
/*-----*/
/* Ejemplo en C# nº 43: */
/* ejemplo43.cs */
/* */
/* Cadenas de texto (2) */
/* */
/* Introduccion a C#, */
/* Nacho Cabanes */
/*-----*/

using System;

public class Ejemplo43
{
    public static void Main()
    {
        string ejemplo = "Hola, que tal estas";

        Console.WriteLine("El texto es: {0}",
            ejemplo);

        Console.WriteLine("La primera letra es: {0}",
            ejemplo[0]);

        Console.WriteLine("Las tres primeras letras son: {0}",
            ejemplo.Substring(0,3));

        Console.WriteLine("La longitud del texto es: {0}",
            ejemplo.Length);

        Console.WriteLine("La posicion de \"que\" es: {0}",
            ejemplo.IndexOf("que"));

        Console.WriteLine("La ultima A esta en la posicion: {0}",
            ejemplo.LastIndexOf("a"));

        Console.WriteLine("En mayúsculas: {0}",
            ejemplo.ToUpper());

        Console.WriteLine("En minúsculas: {0}",
            ejemplo.ToLower());

        Console.WriteLine("Si insertamos \", tio\": {0}",
            ejemplo.Insert(4, ", tio"));
    }
}
```

```

Console.WriteLine("Si borramos las 6 primeras letras: {0}",
    ejemplo.Remove(0, 6));

Console.WriteLine("Si cambiamos ESTAS por ESTAMOS: {0}",
    ejemplo.Replace("estas", "estamos"));

}

```

Y su resultado sería

```

El texto es: Hola, que tal estas
La primera letra es: H
Las tres primeras letras son: Hol
La longitud del texto es: 19
La posicion de "que" es: 6
La ultima A esta en la posicion: 17
En mayúsculas: HOLA, QUE TAL ESTAS
En minúsculas: hola, que tal estas
Si insertamos ", tio": Hola, tio, que tal estas
Si borramos las 6 primeras letras: que tal estas
Si cambiamos ESTAS por ESTAMOS: Hola, que tal estamos

```

Otra posibilidad interesante, aunque un poco más avanzada, es la de descomponer una cadena en trozos, que estén separados por una serie de delimitadores (por ejemplo, espacios o comas). Para ello se puede usar Split, que crea un array a partir de los fragmentos de la cadena, así:

```

/*
-----*/
/* Ejemplo en C# nº 43b: */
/* ejemplo43b.cs */
/*
*/
/* Cadenas de texto (2b) */
/*
*/
/* Introducción a C#, */
/* Nacho Cabanes */
/*
-----*/
using System;

public class Ejemplo43b
{
    public static void Main()
    {
        string ejemplo = "uno,dos,tres,cuatro";
        char [] delimitadores = { ',', '.' };
        int i;

        string [] ejemploPartido = ejemplo.Split(delimitadores);
    }
}

```

```

        for (i=0; i<ejemploPartido.Length; i++)
            Console.WriteLine("Fragmento {0}= {1}",
                i, ejemploPartido[i]);
    }
}

```

Que mostraría en pantalla lo siguiente:

```

Fragmento 0= uno
Fragmento 1= dos
Fragmento 2= tres
Fragmento 3= cuatro

```

4.4.7. Comparación de cadenas

Sabemos comprobar si una cadena tiene exactamente un cierto valor, con el operador de igualdad (==), pero no sabemos comparar qué cadena es "mayor" que otra, algo que es necesario si queremos ordenar textos. El operador "mayor que" (>) que usamos con los números no se puede aplicar directamente a las cadenas. En su lugar, debemos usar "CompareTo", que devolverá un número mayor que 0 si la nuestra cadena es mayor que la que indicamos como parámetro (o un número negativo si nuestra cadena es menor, o 0 si son iguales):

```

if (frase.CompareTo("hola") > 0)
    Console.WriteLine("Es mayor que hola");

```

También podemos comparar sin distinguir entre mayúsculas y minúsculas, usando String.Compare, al que indicamos las dos cadenas y un tercer dato "true" cuando queramos ignorar esa distinción:

```

if (String.Compare(frase, "hola", true) > 0)
    Console.WriteLine("Es mayor que hola (mays o mins)");

```

Un programa completo de prueba podría ser así:

```

/*
 * Ejemplo en C# nº 43c:
 */
/* ejemplo43c.cs */
/*
 */
/* Cadenas de texto (2c) */
/*
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes */
/*
*/

```

```

using System;

public class Ejemplo43c
{
    public static void Main()

```

```

{
    string frase;

    Console.WriteLine("Escriba una palabra");
    frase = Console.ReadLine();

    // Compruebo si es exactamente hola
    if (frase == "hola")
        Console.WriteLine("Ha escrito hola");

    // Compruebo si es mayor o menor
    if (frase.CompareTo("hola") > 0)
        Console.WriteLine("Es mayor que hola");
    else if (frase.CompareTo("hola") < 0)
        Console.WriteLine("Es menor que hola");

    // Comparo sin distinguir mayúsculas ni minúsculas
    bool ignorarMays = true;
    if (String.Compare(frase, "hola", ignorarMays) > 0)
        Console.WriteLine("Es mayor que hola (mays o mins)");
    else if (String.Compare(frase, "hola", ignorarMays) < 0)
        Console.WriteLine("Es menor que hola (mays o mins)");
    else
        Console.WriteLine("Es hola (mays o mins)");
}

}

```

Si tecleamos una palabra como "gol", que comienza por G, que alfabéticamente está antes de la H de "hola", se nos dirá que esa palabra es menor:

```

Escriba una palabra
gol
Es menor que hola
Es menor que hola (mays o mins)

```

Si escribimos "hOLa", que coincide con "hola" salvo por las mayúsculas, una comparación normal nos dirá que es mayor (las mayúsculas se consideran "mayores" que las minúsculas), y una comparación sin considerar mayúsculas o minúsculas nos dirá que coinciden:

```

Escriba una palabra
hOLa
Es mayor que hola
Es hola (mays o mins)

```

4.4.8. Una cadena modificable: **StringBuilder**

Si tenemos la necesidad de poder modificar una cadena letra a letra, no podemos usar un "string" convencional, deberemos recurrir a un "StringBuilder", que sí lo permiten pero son algo más complejos de manejar: hay de reservarles espacio con "new" (igual que hacíamos en

certas ocasiones con los Arrays), y se pueden convertir a una cadena "convencional" usando "ToString":

```
/*
 * Ejemplo en C# nº 44:
 */
/* ejemplo44.cs */
/*
 */
/* Cadena modificable */
/* con "StringBuilder" */
/*
 */
/* Introducción a C#, */
/* Nacho Cabanes */
/*
 */

using System;
using System.Text; // Usaremos un System.Text.StringBuilder

public class Ejemplo44
{
    public static void Main()
    {
        StringBuilder cadenaModificable = new StringBuilder("Hola");
        cadenaModificable[0] = 'M';
        Console.WriteLine("Cadena modificada: {0}",
            cadenaModificable);

        string cadenaNormal;
        cadenaNormal = cadenaModificable.ToString();
        Console.WriteLine("Cadena normal a partir de ella: {0}",
            cadenaNormal);
    }
}
```

Ejercicios propuestos:

- Un programa que pida tu nombre, tu día de nacimiento y tu mes de nacimiento y lo junte todo en una cadena, separando el nombre de la fecha por una coma y el día del mes por una barra inclinada, así: "Juan, nacido el 31/12".
- Crear un juego del ahorcado, en el que un primer usuario introduzca la palabra a adivinar, se muestre esta palabra oculta con guiones (----) y el programa acepte las letras que introduzca el segundo usuario, cambiando los guiones por letras correctas cada vez que acierte (por ejemplo, a---a-t-). La partida terminará cuando se acierte la palabra por completo o el usuario agote sus 8 intentos.

4.4.9. Recorriendo con "foreach"

Existe una construcción parecida a "for", pensada para recorrer ciertas estructuras de datos, como los arrays (y otras que veremos más adelante).

Se usa con el formato "foreach (variable in ConjuntoDeValores)":

```

/*
/* Ejemplo en C# nº 45: */
/* ejemplo45.cs */
/*
/*
/* Ejemplo de "foreach" */
/*
/* Introducción a C#, */
/* Nacho Cabanes */
/*
using System;

public class Ejemplo45
{
    public static void Main()
    {
        int[] diasMes = {31, 28, 21};
        foreach(int dias in diasMes) {
            Console.WriteLine("Días del mes: {0}", dias);
        }

        string[] nombres = {"Alberto", "Andrés", "Antonio"};
        foreach(string nombre in nombres) {
            Console.Write(" {0}", nombre);
        }
        Console.WriteLine();

        string saludo = "Hola";
        foreach(char letra in saludo) {
            Console.Write("{0}-", letra);
        }
        Console.WriteLine();
    }
}

```

4.5 Ejemplo completo

Vamos a hacer un ejemplo completo que use tablas ("arrays"), registros ("struct") y que además manipule cadenas.

La idea va a ser la siguiente: Crearemos un programa que pueda almacenar datos de hasta 1000 ficheros (archivos de ordenador). Para cada fichero, debe guardar los siguientes datos: Nombre del fichero, Tamaño (en KB, un número de 0 a 8.000.000.000). El programa mostrará un menú que permita al usuario las siguientes operaciones:

- 1- Añadir datos de un nuevo fichero
- 2- Mostrar los nombres de todos los ficheros almacenados
- 3- Mostrar ficheros que sean de más de un cierto tamaño (por ejemplo, 2000 KB).
- 4- Ver todos los datos de un cierto fichero (a partir de su nombre)
- 5- Salir de la aplicación (como no usamos ficheros, los datos se perderán).

No debería resultar difícil. Vamos a ver directamente una de las formas en que se podría plantear y luego comentaremos alguna de las mejoras que se podría (incluso se debería) hacer.

Una opción que podemos a tomar para resolver este problema es la de contar el número de fichas que tenemos almacenadas, y así podremos añadir de una en una. Si tenemos 0 fichas, deberemos almacenar la siguiente (la primera) en la posición 0; si tenemos dos fichas, serán la 0 y la 1, luego añadiremos en la posición 2; en general, si tenemos "n" fichas, añadiremos cada nueva ficha en la posición "n". Por otra parte, para revisar todas las fichas, recorreremos desde la posición 0 hasta la n-1, haciendo algo como

```
for (i=0; i<=n-1; i++) { ... más órdenes ...}
```

o bien algo como

```
for (i=0; i<n; i++) { ... más órdenes ...}
```

El resto del programa no es difícil: sabemos leer y comparar textos y números, comprobar varias opciones con "switch", etc. Aun así, haremos una última consideración: hemos limitado el número de fichas a 1000, así que, si nos piden añadir, deberíamos asegurarnos antes de que todavía tenemos hueco disponible.

Con todo esto, nuestro fuente quedaría así:

```
/*
 * Ejemplo en C# nº 46:
 */
/* ejemplo46.cs */
/*
 */
/* Tabla con muchos struct */
/* y menu para manejarla */
/*
 */
/* Introduccion a C#,
/* Nacho Cabanes
/*
 */

using System;

public class Ejemplo46
{

    struct tipoFicha {
        public string nombreFich; /* Nombre del fichero */
        public long tamanyo; /* El tamaño en bytes */
    }

    public static void Main()
    {
        tipoFicha[] fichas /* Los datos en si */
            = new tipoFicha[1000];
        int numeroFichas=0; /* Número de fichas que ya tenemos */
        int i; /* Para bucles */
        int opcion; /* La opcion del menu que elija el usuario */
        string textoBuscar; /* Para cuando preguntaremos al usuario */
        long tamanyoBuscar; /* Para buscar por tamaño */
```

```

do {
    /* Menu principal */
    Console.WriteLine();
    Console.WriteLine("Escoja una opción:");
    Console.WriteLine("1.- Añadir datos de un nuevo fichero");
    Console.WriteLine("2.- Mostrar los nombres de todos los ficheros");
    Console.WriteLine("3.- Mostrar ficheros que sean de mas de un cierto tamaño");
    Console.WriteLine("4.- Ver datos de un fichero");
    Console.WriteLine("5.- Salir");

    opcion = Convert.ToInt32( Console.ReadLine() );

    /* Hacemos una cosa u otra según la opción escogida */
    switch(opcion){
        case 1: /* Añadir un dato nuevo */
            if (numeroFichas < 1000) { /* Si queda hueco */
                Console.WriteLine("Introduce el nombre del fichero: ");
                fichas[numeroFichas].nombreFich = Console.ReadLine();
                Console.WriteLine("Introduce el tamaño en KB: ");
                fichas[numeroFichas].tamanyo = Convert.ToInt32( Console.ReadLine() );
                /* Y ya tenemos una ficha más */
                numeroFichas++;
            } else /* Si no hay hueco para más fichas, avisamos */
                Console.WriteLine("Máximo de fichas alcanzado (1000)!");
            break;
        case 2: /* Mostrar todos */
            for (i=0; i<numeroFichas; i++)
                Console.WriteLine("Nombre: {0}; Tamaño: {1} Kb",
                    fichas[i].nombreFich, fichas[i].tamanyo);
            break;
        case 3: /* Mostrar según el tamaño */
            Console.WriteLine("¿A partir de que tamaño quieres que te muestre?");
            tamanyoBuscar = Convert.ToInt64( Console.ReadLine() );
            for (i=0; i<numeroFichas; i++)
                if (fichas[i].tamanyo >= tamanyoBuscar)
                    Console.WriteLine("Nombre: {0}; Tamaño: {1} Kb",
                        fichas[i].nombreFich, fichas[i].tamanyo);
            break;
        case 4: /* Ver todos los datos (pocos) de un fichero */
            Console.WriteLine("¿De qué fichero quieres ver todos los datos?");
            textoBuscar = Console.ReadLine();
            for (i=0; i<numeroFichas; i++)
                if ( fichas[i].nombreFich == textoBuscar )
                    Console.WriteLine("Nombre: {0}; Tamaño: {1} Kb",
                        fichas[i].nombreFich, fichas[i].tamanyo);
            break;
        case 5: /* Salir: avisamos de que salimos */
            Console.WriteLine("Fin del programa");
            break;
        default: /* Otra opcion: no válida */
            Console.WriteLine("Opción desconocida!");
            break;
    }
} while (opcion != 5); /* Si la opcion es 5, terminamos */
}

```

Funciona, y hace todo lo que tiene que hacer, pero es mejorable. Por supuesto, en un caso real es habitual que cada ficha tenga que guardar más información que sólo esos dos apartados de ejemplo que hemos previsto esta vez. Si nos muestra todos los datos en pantalla y se trata de muchos datos, puede ocurrir que aparezcan en pantalla tan rápido que no nos dé tiempo a leerlos, así que sería deseable que parase cuando se llenase la pantalla de información (por ejemplo, una pausa tras mostrar cada 25 datos). Por supuesto, se nos pueden ocurrir muchas más preguntas que hacerle sobre nuestros datos. Y además, cuando salgamos del programa se borrarán todos los datos que habíamos tecleado, pero eso es lo único "casi inevitable", porque aún no sabemos manejar ficheros.

Ejercicios propuestos:

- Un programa que pida el nombre, el apellido y la edad de una persona, los almacene en un "struct" y luego muestre los tres datos en una misma línea, separados por comas.
- Un programa que pida datos de 8 personas: nombre, dia de nacimiento, mes de nacimiento, y año de nacimiento (que se deben almacenar en una tabla de structs). Después deberá repetir lo siguiente: preguntar un número de mes y mostrar en pantalla los datos de las personas que cumplan los años durante ese mes. Terminará de repetirse cuando se teclee 0 como número de mes.
- Un programa que sea capaz de almacenar los datos de 50 personas: nombre, dirección, teléfono, edad (usando una tabla de structs). Deberá ir pidiendo los datos uno por uno, hasta que un nombre se introduzca vacío (se pulse Intro sin teclear nada). Entonces deberá aparecer un menú que permita:
 - Mostrar la lista de todos los nombres.
 - Mostrar las personas de una cierta edad.
 - Mostrar las personas cuya inicial sea la que el usuario indique.
 - Salir del programa
 (lógicamente, este menú debe repetirse hasta que se escoja la opción de "salir").
- Mejorar la base de datos de ficheros (ejemplo 46) para que no permita introducir tamaños incorrectos (números negativos) ni nombres de fichero vacíos.
- Ampliar la base de datos de ficheros (ejemplo 46) para que incluya una opción de búsqueda parcial, en la que el usuario indique parte del nombre y se muestre todos los ficheros que contienen ese fragmento (usando "IndexOf"). Esta búsqueda no debe distinguir mayúsculas y minúsculas (con la ayuda de ToUpper o ToLower).
- Ampliar el ejercicio anterior (el que permite búsqueda parcial) para que la búsqueda sea incremental: el usuario irá indicando letra a letra el texto que quiere buscar, y se mostrarán todos los datos que lo contienen (por ejemplo, primero los que contienen "j", luego "ju", después "jua" y finalmente "juan").
- Ampliar la base de datos de ficheros (ejemplo 46) para que se pueda borrar un cierto dato (habrá que "mover hacia atrás" todos los datos que había después de ese, y disminuir el contador de la cantidad de datos que tenemos).
- Mejorar la base de datos de ficheros (ejemplo 46) para que se pueda modificar un cierto dato a partir de su número (por ejemplo, el dato número 3). En esa modificación,

se deberá permitir al usuario pulsar Intro sin teclear nada, para indicar que no desea modificar un cierto dato, en vez de reemplazarlo por una cadena vacía.

- Ampliar la base de datos de ficheros (ejemplo 46) para que se permita ordenar los datos por nombre. Para ello, deberás buscar información sobre algún método de ordenación sencillo, como el "método de burbuja" (en el siguiente apartado tienes algunos), y aplicarlo a este caso concreto.

4.6 Ordenaciones simples

Es muy frecuente querer ordenar datos que tenemos en un array. Para conseguirlo, existen varios algoritmos sencillos, que no son especialmente eficientes, pero son fáciles de programar. La falta de eficiencia se refiere a que la mayoría de ellos se basan en dos bucles "for" anidados, de modo que en cada pasada quede ordenado un dato, y se dan tantas pasadas como datos existen, de modo que para un array con 1.000 datos, podrían llegar a tener que hacerse un millón de comparaciones.

Existen ligeras mejoras (por ejemplo, cambiar uno de los "for" por un "while", para no repasar todos los datos si ya estaban parcialmente ordenados), así como métodos claramente más efectivos, pero más difíciles de programar, alguno de los cuales veremos más adelante.

Veremos tres de estos métodos simples de ordenación, primero mirando la apariencia que tiene el algoritmo, y luego juntando los tres en un ejemplo que los pruebe:

Método de burbuja

(Intercambiar cada pareja consecutiva que no esté ordenada)

Para i=1 hasta n-1

 Para j=i+1 hasta n

 Si A[i] > A[j]

 Intercambiar (A[i], A[j])

(Nota: algunos autores hacen el bucle exterior creciente y otros decreciente, así:)

Para i=n descendiendo hasta 2

 Para j=2 hasta i

 Si A[j-1] > A[j]

 Intercambiar (A[j-1], A[j])

Selección directa

(En cada pasada busca el menor, y lo intercambia al final de la pasada)

Para i=1 hasta n-1

 menor = i

 Para j=i+1 hasta n

 Si A[j] < A[menor]

 menor = j

 Si menor <> i

 Intercambiar (A[i], A[menor])

Nota: el símbolo "<>" se suele usar en pseudocódigo para indicar que un dato es distinto de otro, de modo que equivale al "!=" de C#. La penúltima línea en C# saldría a ser algo como "if (menor != i)"

Inserción directa

(Comparar cada elemento con los anteriores -que ya están ordenados- y desplazarlo hasta su posición correcta).

Para $i=2$ hasta n

```
j=i-1
mientras (j>=1) y (A[j] > A[j+1])
    Intercambiar ( A[j], A[j+1])
    j = j - 1
```

(Es mejorable, no intercambiando el dato que se mueve con cada elemento, sino sólo al final de cada pasada, pero no entraremos en más detalles).

Un programa de prueba podría ser:

```
/*
-----*/
/* Ejemplo en C#
/* ordenar.cs */
/*
/* Ordenaciones simples */
/*
/* Introduccion a C#,
/* Nacho Cabanes */
/*
-----*/
using System;

public class Ordenar
{
    public static void Main()
    {
        int[] datos = {5, 3, 14, 20, 8, 9, 1};
        int i,j,datoTemporal;
        int n=7; // Número de datos

        // BURBUJA
        // (Intercambiar cada pareja consecutiva que no esté ordenada)
        // Para i=1 hasta n-1
        //     Para j=i+1 hasta n
        //         Si A[i] > A[j]
        //             Intercambiar ( A[i], A[j])
        Console.WriteLine("Ordenando mediante burbuja... ");
        for(i=0; i < n-1; i++)
        {
            foreach (int dato in datos) // Muestro datos
                Console.Write("{0} ",dato);
```

```

Console.WriteLine();

for(j=i+1; j < n; j++)
{
    if (datos[i] > datos[j])
    {
        datoTemporal = datos[i];
        datos[i] = datos[j];
        datos[j] = datoTemporal;
    }
}
Console.Write("Ordenado:");

foreach (int dato in datos) // Muestro datos finales
    Console.Write("{0} ",dato);
Console.WriteLine();

// SELECCIÓN DIRECTA:
// (En cada pasada busca el menor, y lo intercambia al final de la pasada)
// Para i=1 hasta n-1
//     menor = i
//     Para j=i+1 hasta n
//         Si A[j] < A[menor]
//             menor = j
//     Si menor <> i
//         Intercambiar ( A[i], A[menor] )
Console.WriteLine("\nOrdenando mediante selección directa... ");
int[] datos2 = {5, 3, 14, 20, 8, 9, 1};
for(i=0; i < n-1; i++)
{
    foreach (int dato in datos2) // Muestro datos
        Console.Write("{0} ",dato);
    Console.WriteLine();

    int menor = i;
    for(j=i+1; j < n; j++)
        if (datos2[j] < datos2[menor])
            menor = j;

    if (i != menor)
    {
        datoTemporal = datos2[i];
        datos2[i] = datos2[menor];
        datos2[menor] = datoTemporal;
    }
}
Console.Write("Ordenado:");

foreach (int dato in datos2) // Muestro datos finales
    Console.Write("{0} ",dato);
Console.WriteLine();

// INSERCIÓN DIRECTA:
// (Comparar cada elemento con los anteriores -que ya están ordenados-
// y desplazarlo hasta su posición correcta).

```

```

// Para i=2 hasta n
//   j=i-1
//   mientras (j>=1) y (A[j] > A[j+1])
//     Intercambiar ( A[j], A[j+1])
//     j = j - 1
Console.WriteLine("\nOrdenando mediante burbuja...");
int[] datos3 = {5, 3, 14, 20, 8, 9, 1};
for(i=1; i < n; i++)
{
    foreach (int dato in datos3) // Muestro datos
        Console.Write("{0} ",dato);
    Console.WriteLine();

    j = i-1;
    while ((j>=0) && (datos3[j] > datos3[j+1]))
    {
        datoTemporal = datos3[j];
        datos3[j] = datos3[j+1];
        datos3[j+1] = datoTemporal;
        j--;
    }
}
Console.WriteLine("Ordenado:");

foreach (int dato in datos3) // Muestro datos finales
    Console.Write("{0} ",dato);
Console.WriteLine();
}

```

Y su resultado sería:

```

Ordenando mediante burbuja...
5 3 14 20 8 9 1
1 5 14 20 8 9 3
1 3 14 20 8 9 5
1 3 5 20 14 9 8
1 3 5 8 20 14 9
1 3 5 8 9 20 14
Ordenado:1 3 5 8 9 14 20

```

```

Ordenando mediante selección directa...
5 3 14 20 8 9 1
1 3 14 20 8 9 5
1 3 14 20 8 9 5
1 3 5 20 8 9 14
1 3 5 8 20 9 14
1 3 5 8 9 20 14
Ordenado:1 3 5 8 9 14 20

```

```

Ordenando mediante inserción directa...
5 3 14 20 8 9 1

```

```
3 5 14 20 8 9 1  
3 5 14 20 8 9 1  
3 5 14 20 8 9 1  
3 5 8 14 20 9 1  
3 5 8 9 14 20 1  
Ordenado:1 3 5 8 9 14 20
```

5. Introducción a las funciones

5.1. Diseño modular de programas: Descomposición modular

Hasta ahora hemos estado pensando los pasos que deberíamos dar para resolver un cierto problema, y hemos creado programas a partir de cada uno de esos pasos. Esto es razonable cuando los problemas son sencillos, pero puede no ser la mejor forma de actuar cuando se trata de algo más complicado.

A partir de ahora vamos a empezar a intentar descomponer los problemas en trozos más pequeños, que sean más fáciles de resolver. Esto nos puede suponer varias ventajas:

- Cada "trozo de programa" independiente será más fácil de programar, al realizar una función breve y concreta.
- El "programa principal" será más fácil de leer, porque no necesitará contener todos los detalles de cómo se hace cada cosa.
- Podremos repartir el trabajo, para que cada persona se encargue de realizar un "trozo de programa", y finalmente se integrará el trabajo individual de cada persona.

Esos "trozos" de programa son lo que suele llamar "subrutinas", "procedimientos" o "funciones". En el lenguaje C y sus derivados, el nombre que más se usa es el de **funciones**.

5.2. Conceptos básicos sobre funciones

En C#, al igual que en C y los demás lenguajes derivados de él, todos los "trozos de programa" son funciones, incluyendo el propio cuerpo de programa, **Main**. De hecho, la forma básica de **definir** una función será indicando su nombre seguido de unos paréntesis vacíos, como hacíamos con "Main", y precediéndolo por ciertas palabras reservadas, como "public static void", cuyo significado iremos viendo muy pronto. Después, entre llaves indicaremos todos los pasos que queremos que dé ese "trozo de programa".

Por ejemplo, podríamos crear una función llamada "saludar", que escribiera varios mensajes en la pantalla:

```
public static void saludar()
{
    Console.WriteLine("Bienvenido al programa");
    Console.WriteLine(" de ejemplo");
    Console.WriteLine("Espero que estés bien");
}
```

Ahora desde dentro del cuerpo de nuestro programa, podríamos **"llamar"** a esa función:

```
public static void Main()
```

```
{
    saludar();
    ...
}
```

Así conseguimos que nuestro programa principal sea más fácil de leer.

Un detalle importante: tanto la función habitual "Main" como la nueva función "Saludar" serían parte de nuestra "class", es decir, el fuente completo sería así:

```
/*
 *----- Ejemplo en C# nº 47:
 *----- ejemplo47.cs
 *----- Funcion "saludar"
 *----- Introduccion a C#,
 *----- Nacho Cabanes
 *----- */

using System;

public class Ejemplo47
{
    public static void Saludar()
    {
        Console.WriteLine("Bienvenido al programa");
        Console.WriteLine(" de ejemplo");
        Console.WriteLine("Espero que estés bien");
    }

    public static void Main()
    {
        Saludar();
        Console.WriteLine("Nada más por hoy...");
    }
}
```

Como ejemplo más detallado, la parte principal de una agenda podría ser simplemente:

```
leerDatosDeFichero();
do {
    mostrarMenu();
    pedirOpcion();
    switch( opcion ) {
        case 1: buscarDatos(); break;
        case 2: modificarDatos(); break;
        case 3: anadirDatos(); break;
    }
    ...
}
```

5.3. Parámetros de una función

Es muy frecuente que nos interese además indicarle a nuestra función ciertos datos especiales con los que queremos que trabaje. Por ejemplo, si escribimos en pantalla números reales con frecuencia, nos puede resultar útil que nos los muestre con el formato que nos interese. Lo podríamos hacer así:

```
public static void escribeNumeroReal( float n )
{
    Console.WriteLine( n.ToString("#.###") );
}
```

Y esta función se podría usar desde el cuerpo de nuestro programa así:

```
escribeNumeroReal(2.3f);
```

(recordemos que el sufijo "f" es para indicar al compilador que trate ese número como un "float", porque de lo contrario, al ver que tiene cifras decimales, lo tomaría como "double", que permite mayor precisión... pero a cambio nosotros tendríamos un mensaje de error en nuestro programa, diciendo que estamos dando un dato "double" a una función que espera un "float").

El programa completo podría quedar así:

```
/*
 *----- Ejemplo en C# nº 48:
 *----- ejemplo48.cs
 *----- Funcion
 *----- "escribeNumeroReal"
 *----- Introduccion a C#,
 *----- Nacho Cabanes
 *----- */

using System;

public class Ejemplo48
{
    public static void escribeNumeroReal( float n )
    {
        Console.WriteLine( n.ToString("#.###") );
    }

    public static void Main()
    {
        float x;

        x= 5.1f;
        Console.WriteLine("El primer numero real es: ");
        escribeNumeroReal(x);
        Console.WriteLine(" y otro distinto es: ");
        escribeNumeroReal(2.3f);
    }
}
```

```
    }
```

Estos datos adicionales que indicamos a la función es lo que llamaremos sus "parámetros". Como se ve en el ejemplo, tenemos que indicar un nombre para cada parámetro (puede haber varios) y el tipo de datos que corresponde a ese parámetro. Si hay más de un parámetro, deberemos indicar el tipo y el nombre para cada uno de ellos, y separarlos entre comas:

```
public static void sumar ( int x, int y ) {
    ...
}
```

5.4. Valor devuelto por una función. El valor "void".

Cuando queremos dejar claro que una función no tiene que devolver ningún valor, podemos hacerlo indicando al principio que el tipo de datos va a ser "void" (nulo), como hacíamos hasta ahora con "Main" y como hicimos con nuestra función "saludar".

Pero eso no es lo que ocurre con las funciones matemáticas que estamos acostumbrados a manejar: sí devuelven un valor, el resultado de una operación.

De igual modo, para nosotros también será habitual que queramos que nuestra función realice una serie de cálculos y nos "devuelva" (return, en inglés) el resultado de esos cálculos, para poderlo usar desde cualquier otra parte de nuestro programa. Por ejemplo, podríamos crear una función para elevar un número entero al cuadrado así:

```
public static int cuadrado ( int n )
{
    return n*n;
}
```

y podríamos usar el resultado de esa función como si se tratara de un número o de una variable, así:

```
resultado = cuadrado( 5 );
```

Un programa más detallado de ejemplo podría ser:

```
/*-----*/
/* Ejemplo en C# nº 49:      */
/* ejemplo49.cs               */
/*                           */
/* Funcion "cuadrado"        */
/*                           */
/* Introduccion a C#,        */
/* Nacho Cabanes             */
/*-----*/
using System;

public class Ejemplo49
{
```

```

public static int cuadrado ( int n )
{
    return n*n;
}

public static void Main()
{
    int numero;
    int resultado;

    numero= 5;
    resultado = cuadrado(numero);
    Console.WriteLine("El cuadrado del numero {0} es {1}",
                      numero, resultado);
    Console.WriteLine(" y el de 3 es {0}", cuadrado(3));
}

```

Podemos hacer una función que nos diga cual es el mayor de dos números reales así:

```

public static float mayor ( float n1, float n2 )
{
    if (n1 > n2)
        return n1;
    else
        return n2;
}

```

Ejercicios propuestos:

- Crear una función que borre la pantalla dibujando 25 líneas en blanco. No debe devolver ningún valor.
- Crear una función que calcule el cubo de un número real (float). El resultado deberá ser otro número real. Probar esta función para calcular el cubo de 3.2 y el de 5.
- Crear una función que calcule cual es el menor de dos números enteros. El resultado será otro número entero.
- Crear una función llamada "signo", que reciba un número real, y devuelva un número entero con el valor: -1 si el número es negativo, 1 si es positivo o 0 si es cero.
- Crear una función que devuelva la primera letra de una cadena de texto. Probar esta función para calcular la primera letra de la frase "Hola".
- Crear una función que devuelva la última letra de una cadena de texto. Probar esta función para calcular la última letra de la frase "Hola".
- Crear una función que reciba un número y muestre en pantalla el perímetro y la superficie de un cuadrado que tenga como lado el número que se ha indicado como parámetro.

5.5. Variables locales y variables globales

Hasta ahora, hemos declarado las variables dentro de "Main". Ahora nuestros programas tienen varios "bloques", así que se comportarán de forma distinta según donde declaremos las variables.

Las variables se pueden declarar dentro de un bloque (una función), y entonces sólo ese bloque las conocerá, no se podrán usar desde ningún otro bloque del programa. Es lo que llamaremos "variables **locales**".

Por el contrario, si declaramos una variable al comienzo del programa, fuera de todos los "bloques" de programa, será una "**variable global**", a la que se podrá acceder desde cualquier parte.

Vamos a verlo con un ejemplo. Crearemos una función que calcule la potencia de un número entero (un número elevado a otro), y el cuerpo del programa que la use.

La forma de conseguir elevar un número a otro será a base de multiplicaciones, es decir:

$$3 \text{ elevado a } 5 = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$$

(multiplicamos 5 veces el 3 por sí mismo). En general, como nos pueden pedir cosas como "6 elevado a 100" (o en general números que pueden ser grandes), usaremos la orden "for" para multiplicar tantas veces como haga falta:

```
/*
 * Ejemplo en C# nº 50:
 */
/* ejemplo50.cs */
/*
 */
/* Ejemplo de función con */
/* variables locales */
/*
 */
/* Introducción a C#,
/* Nacho Cabanes */
/*
 */

using System;

public class Ejemplo50
{
    public static int potencia(int nBase, int nExponente)
    {
        int temporal = 1;           /* Valor que voy hallando */
        int i;                     /* Para bucles */

        for(i=1; i<=nExponente; i++) /* Multiplico "n" veces */
            temporal *= nBase;      /* Y calculo el valor temporal */

        return temporal;           /* Tras las multiplicaciones,
                                     /* obtengo el valor que buscaba */
    }

    public static void Main()
    {
```

```

int num1, num2;

Console.WriteLine("Introduzca la base: ");
num1 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Introduzca el exponente: ");
num2 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("{0} elevado a {1} vale {2}",
    num1, num2, potencia(num1,num2));
}

}

```

En este caso, las variables "temporal" e "i" son locales a la función "potencia": para "Main" no existen. Si en "Main" intentáramos hacer `i=5;` obtendríamos un mensaje de error.

De igual modo, "num1" y "num2" son locales para "main": desde la función "potencia" no podemos acceder a su valor (ni para leerlo ni para modificarlo), sólo desde "main".

En general, deberemos intentar que la mayor cantidad de variables posible sean locales (lo ideal sería que todas lo fueran). Así hacemos que cada parte del programa trabaje con sus propios datos, y ayudamos a evitar que un error en un trozo de programa pueda afectar al resto. La forma correcta de pasar datos entre distintos trozos de programa es usando los parámetros de cada función, como en el anterior ejemplo.

Ejercicios propuestos:

- Crear una función "pedirEntero", que reciba como parámetros el texto que se debe mostrar en pantalla, el valor mínimo aceptable y el valor máximo aceptable. Deberá pedir al usuario que introduzca el valor tantas veces como sea necesario, volvérsele a pedir en caso de error, y devolver un valor correcto. Probarlo con un programa que pida al usuario un año entre 1800 y 2100.
- Crear una función "escribirTablaMultiplicar", que reciba como parámetro un número entero, y escriba la tabla de multiplicar de ese número (por ejemplo, para el 3 deberá llegar desde $3 \times 0 = 0$ hasta $3 \times 10 = 30$).
- Crear una función "esPrimo", que reciba un número y devuelva el valor booleano "true" si es un número primo o "false" en caso contrario.
- Crear una función que reciba una cadena y una letra, y devuelva la cantidad de veces que dicha letra aparece en la cadena. Por ejemplo, si la cadena es "Barcelona" y la letra es 'a', debería devolver 2 (aparece 2 veces).
- Crear una función que reciba un numero cualquiera y que devuelva como resultado la suma de sus dígitos. Por ejemplo, si el número fuera 123 la suma sería 6.
- Crear una función que reciba una letra y un número, y escriba un "triángulo" formado por esa letra, que tenga como anchura inicial la que se ha indicado. Por ejemplo, si la letra es '*' y la anchura es 4, debería escribir

**
*

5.6. Los conflictos de nombres en las variables

¿Qué ocurre si damos el mismo nombre a dos variables locales? Vamos a comprobarlo con un ejemplo:

```
/*
/* Ejemplo en C# nº 51: */
/* ejemplo51.cs */
/*
/*
/* Dos variables locales */
/* con el mismo nombre */
/*
/*
/* Introducción a C#, */
/* Nacho Cabanes */
/*
using System;

public class Ejemplo51
{
    public static void cambiaN()
    {
        int n = 7;
        n++;
    }

    public static void Main()
    {
        int n = 5;
        Console.WriteLine("n vale {0}", n);
        cambiaN();
        Console.WriteLine("Ahora n vale {0}", n);
    }
}
```

El resultado de este programa es:

```
n vale 5
Ahora n vale 5
```

¿Por qué? Sencillo: tenemos una variable local dentro de "cambiaN" y otra dentro de "main". El hecho de que las dos tengan el mismo nombre no afecta al funcionamiento del programa, siguen siendo distintas.

Si la variable es "global", declarada fuera de estas funciones, sí será accesible por todas ellas:

```
/*
/* Ejemplo en C# nº 52: */
/* ejemplo52.cs */
/*
/*
/* Una variable global */
/*
```

```

/*
 * Introduccion a C#,
 * Nacho Cabanes
 */
using System;

public class Ejemplo52
{
    static int n = 7;

    public static void cambiaN() {
        n++;
    }

    public static void Main()
    {
        Console.WriteLine("n vale {0}", n);
        cambiaN();
        Console.WriteLine("Ahora n vale {0}", n);
    }
}

```

Dentro de poco, hablaremos de por qué cada uno de los bloques de nuestro programa, e incluso las "variables globales", tienen delante la palabra "static". Será cuando tratemos la "Programación Orientada a Objetos", en el próximo tema.

5.7. Modificando parámetros

Podemos modificar el valor de un dato que recibamos como parámetro, pero posiblemente el resultado no será el que esperamos. Vamos a verlo con un ejemplo:

```

/*
 * Ejemplo en C# nº 53:      */
/* ejemplo53.cs               */
/*                                */
/* Modificar una variable      */
/* recibida como parámetro     */
/*                                */
/* Introduccion a C#,          */
/* Nacho Cabanes               */
*/
using System;

public class Ejemplo53
{
    public static void duplica(int x) {
        Console.WriteLine(" El valor recibido vale {0}", x);
        x = x * 2;
        Console.WriteLine(" y ahora vale {0}", x);
    }
}

```

```

public static void Main()
{
    int n = 5;
    Console.WriteLine("n vale {0}", n);
    duplica(n);
    Console.WriteLine("Ahora n vale {0}", n);
}

}

```

El resultado de este programa será:

```

n vale 5
El valor recibido vale 5
y ahora vale 10
Ahora n vale 5

```

Vemos que al salir de la función, los cambios que hagamos a esa variable que se ha recibido como parámetro no se conservan.

Esto se debe a que, si no indicamos otra cosa, los parámetros **"se pasan por valor"**, es decir, la función no recibe los datos originales, sino una copia de ellos. Si modificamos algo, estamos cambiando una copia de los datos originales, no dichos datos.

Si queremos que los cambios se conserven, basta con hacer un pequeño cambio: indicar que la variable se va a pasar **"por referencia"**, lo que se indica usando la palabra "ref", tanto en la declaración de la función como en la llamada, así:

```

/*
 * Ejemplo en C# nº 54:
 * ejemplo54.cs
 *
 * Modificar una variable
 * recibida como parámetro
 *
 * Introducción a C#
 * Nacho Cabanes
 */

using System;

public class Ejemplo54
{

    public static void duplica(ref int x) {
        Console.WriteLine("El valor recibido vale {0}", x);
        x = x * 2;
        Console.WriteLine("y ahora vale {0}", x);
    }

    public static void Main()
{
    int n = 5;
}

```

```

Console.WriteLine("n vale {0}", n);
duplica(ref n);
Console.WriteLine("Ahora n vale {0}", n);
}

}

```

En este caso sí se modifica la variable n:

```

n vale 5
El valor recibido vale 5
y ahora vale 10
Ahora n vale 10

```

El hecho de poder modificar valores que se reciban como parámetros abre una posibilidad que no se podría conseguir de otra forma: con "return" sólo se puede devolver un valor de una función, pero con parámetros pasados por referencia podríamos devolver más de un dato. Por ejemplo, podríamos crear una función que intercambiara los valores de dos variables:

```
public static void intercambia(ref int x, ref int y)
```

La posibilidad de pasar parámetros por valor y por referencia existe en la mayoría de lenguajes de programación. En el caso de C# existe alguna posibilidad adicional que no existe en otros lenguajes, como los "parámetros de salida". Las veremos más adelante.

Ejercicios propuestos:

- Crear una función "intercambia", que intercambie el valor de los dos números enteros que se le indiquen como parámetro.
- Crear una función "iniciales", que reciba una cadena como "Nacho Cabanes" y devuelva las letras N y C (primera letra, y letra situada tras el primer espacio), usando parámetros por referencia.

5.8. El orden no importa

En algunos lenguajes, una función debe estar declarada antes de usarse. Esto no es necesario en C#. Por ejemplo, podríamos rescribir el fuente anterior, de modo que "Main" aparezca en primer lugar y "duplica" aparezca después, y seguiría compilando y funcionando igual:

```

/*-----*/
/* Ejemplo en C# nº 55:      */
/* ejemplo55.cs               */
/*                           */
/* Función tras Main          */
/*                           */
/* Introducción a C#,          */
/* Nacho Cabanes              */
/*-----*/

```

```
using System;
```

```

public class Ejemplo55
{
    public static void Main()
    {
        int n = 5;
        Console.WriteLine("n vale {0}", n);
        duplica(ref n);
        Console.WriteLine("Ahora n vale {0}", n);
    }

    public static void duplica(ref int x) {
        Console.WriteLine("El valor recibido vale {0}", x);
        x = x * 2;
        Console.WriteLine("y ahora vale {0}", x);
    }
}

```

5.9. Algunas funciones útiles

5.9.1. Números aleatorios

En un programa de gestión o una utilidad que nos ayuda a administrar un sistema, no es habitual que podamos permitir que las cosas ocurran al azar. Pero los juegos se encuentran muchas veces entre los ejercicios de programación más completos, y para un juego sí suele ser conveniente que haya algo de azar, para que una partida no sea exactamente igual a la anterior.

Generar números al azar ("números aleatorios") usando C# no es difícil: debemos crear un objeto de tipo "Random", y luego llamaremos a "Next" para obtener valores entre dos extremos:

```

// Creamos un objeto random
Random r = new Random();

// Generamos un número entre dos valores dados
int aleatorio = r.Next(1, 100);

```

Podemos hacer que sea realmente un poco más aleatorio si en la primera orden le indicamos que tome como semilla el instante actual:

```
Random r = new Random(DateTime.Now.Millisecond);
```

De hecho, una forma muy simple de obtener un número "casi al azar" entre 0 y 999 es tomar las milésimas de segundo de la hora actual:

```
int falsoAleatorio = DateTime.Now.Millisecond;
```

Vamos a ver un ejemplo, que muestre en pantalla un número al azar entre 1 y 10:

```
/*-----*/
/* Ejemplo en C# nº 56:      */
/* ejemplo56.cs            */
/*                         */
/* Números al azar          */
/*                         */
/* Introducción a C#,       */
/* Nacho Cabanes           */
/*-----*/
using System;

public class Ejemplo56
{
    public static void Main()
    {
        Random r = new Random(DateTime.Now.Millisecond);
        int aleatorio = r.Next(1, 10);
        Console.WriteLine("Un número entre 1 y 10: {0}",
                          aleatorio);
    }
}
```

Ejercicios propuestos:

- Crear un programa que genere un número al azar entre 1 y 100. El usuario tendrá 6 oportunidades para acertarlo.
- Mejorar el programa del ahorcado propuesto en el apartado 4.4.8, para que la palabra a adivinar no sea tecleado por un segundo usuario, sino que se escoja al azar de un "array" de palabras prefijadas (por ejemplo, nombres de ciudades).

5.9.2. Funciones matemáticas

En C# tenemos muchas funciones matemáticas predefinidas, como:

- Abs(x): Valor absoluto
- Acos(x): Arco coseno
- Asin(x): Arco seno
- Atan(x): Arco tangente
- Atan2(y,x): Arco tangente de y/x (por si x o y son 0)
- Ceiling(x): El valor entero superior a x y más cercano a él
- Cos(x): Coseno
- Cosh(x): Coseno hiperbólico
- Exp(x): Exponencial de x (e elevado a x)
- Floor(x): El mayor valor entero que es menor que x
- Log(x): Logaritmo natural (o neperiano, en base "e")
- Log10(x): Logaritmo en base 10
- Pow(x,y): x elevado a y
- Round(x, cifras): Redondea un número

- `Sin(x)`: Seno
- `Sinh(x)`: Seno hiperbólico
- `Sqrt(x)`: Raíz cuadrada
- `Tan(x)`: Tangente
- `Tanh(x)`: Tangente hiperbólica

(casi todos ellos usan parámetros X e Y de tipo "double")

y una serie de constantes como

`E`, el número "e", con un valor de 2.71828...
`PI`, el número "Pi", 3.14159...

Todas ellas se usan precedidas por "Math."

La mayoría de ellas son específicas para ciertos problemas matemáticos, especialmente si interviene la trigonometría o si hay que usar logaritmos o exponenciales. Pero vamos a destacar las que sí pueden resultar útiles en situaciones más variadas:

La raíz cuadrada de 4 se calcularía haciendo `x = Math.Sqrt(4);`

La potencia: para elevar 2 al cubo haríamos `y = Math.Pow(2, 3);`

El valor absoluto: para trabajar sólo con números positivos usaríamos `n = Math.Abs(x);`

Ejercicios propuestos:

- Crear un programa que halle cualquier raíz de un número. El usuario deberá indicar el número (por ejemplo, 2) y el índice de la raíz (por ejemplo, 3 para la raíz cúbica). Pista: hallar la raíz cúbica de 2 es lo mismo que elevar 2 a 1/3.
- Crear un programa que resuelva ecuaciones de segundo grado, del tipo $ax^2 + bx + c = 0$. El usuario deberá introducir los valores de a, b y c. Se deberá crear una función "raicesSegundoGrado", que recibirá como parámetros los coeficientes a, b y c, así como las soluciones x_1 y x_2 (por referencia). Deberá devolver los valores de las dos soluciones x_1 y x_2 . Si alguna solución no existe, se devolverá como valor 100.000 para esa solución. Pista: la solución se calcula con

$$x = -b \pm \sqrt{b^2 - 4 \cdot a \cdot c} / 2 \cdot a$$

5.9.3. Pero hay muchas más funciones...

Pero en C# hay muchas más funciones de lo que parece. De hecho, salvo algunas palabras reservadas (`int`, `float`, `string`, `if`, `switch`, `for`, `do`, `while`...), gran parte de lo que hasta ahora hemos llamado "órdenes", son realmente "funciones", como `Console.ReadLine` o `Console.WriteLine`. Nos iremos encontrando con otras funciones a medida que avancemos.

5.10. Recursividad

Una función recursiva es aquella que se define a partir de ella misma. Dentro de las matemáticas tenemos varios ejemplos. Uno clásico es el "factorial de un número":

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

(por ejemplo, el factorial de 4 es $4 \cdot 3 \cdot 2 \cdot 1 = 24$)

Si pensamos que el factorial de $n-1$ es

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

Entonces podemos escribir el factorial de un número a partir del factorial del siguiente número:

$$n! = n \cdot (n-1)!$$

Esta es la definición recursiva del factorial, ni más ni menos. Esto, programando, se haría:

```
/*
 *----- Ejemplo en C# nº 57:
 *----- ejemplo57.cs
 */
/*
 *----- Funciones recursivas:
 *----- factorial
 */
/*
 *----- Introducción a C#,
 *----- Nacho Cabanes
 */
using System;

public class Ejemplo57
{
    public static long fact(int n) {
        if (n==1) // Aseguramos que termine
            return 1;
        return n * fact (n-1); // Si no es 1, sigue la recursión
    }

    public static void Main()
    {
        int num;
        Console.WriteLine("Introduzca un número entero: ");
        num = Convert.ToInt32(System.Console.ReadLine());
        Console.WriteLine("Su factorial es: {0}", fact(num));
    }
}
```

Dos consideraciones importantes:

- Atención a la primera parte de la función recursiva: es MUY IMPORTANTE comprobar que hay **salida** de la función, para que nuestro programa no se quede dando vueltas todo el tiempo y deje el ordenador (o la tarea actual) "colgado".
- Los factoriales **crecen rápidamente**, así que no conviene poner números grandes: el factorial de 16 es 2.004.189.184, luego a partir de 17 podemos obtener resultados erróneos, si usamos números enteros "normales".

¿Qué utilidad tiene esto? Pues más de la que parece: muchos problemas complicados se pueden expresar a partir de otro más sencillo. En muchos de esos casos, ese problema se podrá expresar de forma recursiva. Más adelante veremos algún otro ejemplo.

Ejercicios propuestos:

- Crear una función que calcule el valor de elevar un número entero a otro número entero (por ejemplo, 5 elevado a 3 = $5^3 = 5 \cdot 5 \cdot 5 = 125$). Esta función se debe crear de forma recursiva.
- Como alternativa, crear una función que calcule el valor de elevar un número entero a otro número entero de forma NO recursiva (lo que llamaremos "de forma iterativa"), usando la orden "for".
- Crear un programa que emplee recursividad para calcular un número de la serie Fibonacci (en la que los dos primeros elementos valen 1, y para los restantes, cada elemento es la suma de los dos anteriores).
- Crear un programa que emplee recursividad para calcular la suma de los elementos de un vector.
- Crear un programa que emplee recursividad para calcular el mayor de los elementos de un vector.
- Crear un programa que emplee recursividad para dar la vuelta a una cadena de caracteres (por ejemplo, a partir de "Hola" devolvería "aloH").
- Crear, tanto de forma recursiva como de forma iterativa, una función diga si una cadena de caracteres es simétrica (un palíndromo). Por ejemplo, "DABALEARROZALAZORRAELABAD" es un palíndromo.
- Crear un programa que encuentre el máximo común divisor de dos números usando el algoritmo de Euclides: Dados dos números enteros positivos m y n, tal que $m > n$, para encontrar su máximo común divisor, es decir, el mayor entero positivo que divide a ambos: - Dividir m por n para obtener el resto r ($0 \leq r < n$) ; - Si $r = 0$, el MCD es n.; - Si no, el máximo común divisor es $MCD(n,r)$.

5.11. Parámetros y valor de retorno de "Main"

Es muy frecuente que un programa que usamos desde la "línea de comandos" tenga ciertas opciones que le indicamos como argumentos. Por ejemplo, bajo Linux o cualquier otro sistema operativo de la familia Unix, podemos ver la lista detallada de ficheros que terminan en .c haciendo

```
ls -l *.c
```

En este caso, la orden sería "ls", y las dos opciones (argumentos o parámetros) que le indicamos son "-l" y "*.c".

La orden equivalente en MsDos y en el intérprete de comandos de Windows sería

```
dir *.c
```

Ahora la orden sería "dir", y el parámetro es "*.c".

Pues bien, estas opciones que se le pasan al programa se pueden leer desde C#. Se hace indicando un parámetro especial en Main, un array de strings:

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

Para conocer esos parámetros lo haríamos de la misma forma que se recorre habitualmente un array cuyo tamaño no conocemos: con un "for" que termine en la longitud ("Length") del array:

```
for (int i = 0; i < args.Length; i++)
{
    System.Console.WriteLine("El parametro {0} es: {1}",
        i, args[i]);
}
```

Por otra parte, si queremos que nuestro programa **se interrumpa** en un cierto punto, podemos usar la orden "Environment.Exit". Su manejo habitual es algo como

```
Environment.Exit(1);
```

Es decir, entre paréntesis indicamos un cierto código, que suele ser (por convenio) un 0 si no ha habido ningún error, u otro código distinto en caso de que sí exista algún error.

Este valor se podría comprobar desde el sistema operativo. Por ejemplo, en MsDOS y Windows se lee con "IF ERRORLEVEL", así:

```
IF ERRORLEVEL 1 ECHO Ha habido un error en el programa
```

Una forma alternativa de que "Main" indique errores al sistema operativo es no declarándolo como "void", sino como "int", y empleando entonces la orden "return" cuando nos interese:

```
public static int Main(string[] args)
{
    ...
    return 1;
}
```

Un ejemplo que pusiera todo esto en prueba podría ser:

```
/*-----*/
/* Ejemplo en C# nº 58:      */
/* ejemplo58.cs               */
/*                           */
/* Parámetros y valor de     */
/* retorno de "Main"          */
/*                           */
/* Introducción a C#,         */
/* Nacho Cabanes             */
/*-----*/
using System;

public class Ejemplo58
{

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

```
{  
    Console.WriteLine("Parámetros: {0}", args.Length);  
  
    for (int i = 0; i < args.Length; i++)  
    {  
        Console.WriteLine("El parámetro {0} es: {1}",  
            i, args[i]);  
    }  
  
    if (args.Length == 0)  
    {  
        Console.WriteLine("Escriba algún parámetro!");  
        Environment.Exit(1);  
    }  
  
    return 0;  
}  
}
```

Ejercicios propuestos:

- Crear un programa llamado "suma", que calcule (y muestre) la suma de dos números que se le indiquen como parámetro. Por ejemplo, si se teclea "suma 2 3" deberá responder "5", y si se teclea "suma 2" deberá responder "no hay suficientes datos y devolver un código de error 1.
- Crear una calculadora básica, llamada "calcula", que deberá sumar, restar, multiplicar o dividir los dos números que se le indiquen como parámetros. Ejemplos de su uso sería "calcula 2 + 3" o "calcula 5 * 60".

6. Programación orientada a objetos

6.1. ¿Por qué los objetos?

Hasta ahora hemos estado "cuadriculando" todo para obtener algoritmos: tratábamos de convertir cualquier cosa en una función que pudiéramos emplear en nuestros programas. Cuando teníamos claros los pasos que había que dar, buscábamos las variables necesarias para dar esos pasos.

Pero no todo lo que nos rodea es tan fácil de cuadricular. Supongamos por ejemplo que tenemos que introducir datos sobre una puerta en nuestro programa. ¿Nos limitamos a programar los procedimientos AbrirPuerta y CerrarPuerta? Al menos, deberíamos ir a la zona de declaración de variables, y allí guardaríamos otras datos como su tamaño, color, etc.

No está mal, pero es "antinatural": una puerta es un conjunto: no podemos separar su color de su tamaño, o de la forma en que debemos abrirla o cerrarla. Sus características son tanto las físicas (lo que hasta ahora llamábamos variables) como sus comportamientos en distintas circunstancias (lo que para nosotros eran las funciones). Todo ello va unido, formando un **"objeto"**.

Por otra parte, si tenemos que explicar a alguien lo que es el portón de un garaje, y ese alguien no lo ha visto nunca, pero conoce cómo es la puerta de su casa, le podemos decir "se parece a una puerta de una casa, pero es más grande para que quepan los coches, está hecha de metal en vez de madera...". Es decir, podemos describir unos objetos a partir de lo que conocemos de otros.

Finalmente, conviene recordar que "abrir" no se refiere sólo a una puerta. También podemos hablar de abrir una ventana o un libro, por ejemplo.

Con esto, hemos comentado casi sin saberlo las tres características más importantes de la Programación Orientada a Objetos (OOP):

- **Encapsulación:** No podemos separar los comportamientos de las características de un objeto. Los comportamientos serán funciones, que en OOP llamaremos **métodos**. Las características de un objeto serán variables, como las que hemos usado siempre (las llamaremos **atributos**). La apariencia de un objeto en C#, como veremos un poco más adelante, recordará a un registro o "struct".
- **Herencia:** Unos objetos pueden heredar métodos y datos de otros. Esto hace más fácil definir objetos nuevos a partir de otros que ya teníamos anteriormente (como ocurría con el portón y la puerta) y facilitará la reescritura de los programas, pudiendo aprovechar buena parte de los anteriores... si están bien diseñados.
- **Polimorfismo:** Un mismo nombre de un método puede hacer referencia a comportamientos distintos (como abrir una puerta o un libro). Igual ocurre para los datos: el peso de una puerta y el de un portón los podemos llamar de igual forma, pero obviamente no valdrán lo mismo.

Otro concepto importante es el de "clase": **Una clase** es un conjunto de objetos que tienen características comunes. Por ejemplo, tanto mi puerta como la de mi vecino son puertas, es decir, ambas son objetos que pertenecen a la clase "puerta". De igual modo, tanto un Ford Focus como un Honda Civic o un Toyota Corolla son objetos concretos que pertenecen a la clase "coche".

6.2. Objetos y clases en C#

Vamos con los detalles. Las **clases en C#** se definen de forma parecida a los registros (struct), sólo que ahora también incluirán funciones. Así, la clase "Puerta" que mencionábamos antes se podría declarar así:

```
public class Puerta
{
    int ancho;      // Ancho en centimetros
    int alto;       // Alto en centimetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
} // Final de la clase Puerta
```

Como se puede observar, los objetos de la clase "Puerta" tendrán un ancho, un alto, un color, y un estado (abierta o no abierta), y además se podrán abrir o cerrar (y además, nos pueden "mostrar su estado, para comprobar que todo funciona correctamente).

Para declarar estos objetos que pertenecen a la clase "Puerta", usaremos la palabra "new", igual que hacíamos con los "arrays":

```
Puerta p = new Puerta();
p.Abrir();
p.MostrarEstado();
```

Vamos a completar un programa de prueba que use un objeto de esta clase (una "Puerta"):

```

/*
/*----- Ejemplo en C# nº 59: */
/* ejemplo59.cs */
/*
/*
/* Primer ejemplo de clases */
/*
/* Introduccion a C#, */
/*
/* Nacho Cabanes */
/*-----*/
using System;

public class Puerta
{
    int ancho;      // Ancho en centimetros
    int alto;       // Alto en centimetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
}

// Final de la clase Puerta

public class Ejemplo59
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine("\nVamos a abrir...");
        p.Abrir();
        p.MostrarEstado();
    }
}

```

Este fuente ya no contiene una única clase (class), como todos nuestros ejemplos anteriores, sino dos clases distintas:

- La clase "Puerta", que son los nuevos objetos con los que vamos a practicar.
- La clase "Ejemplo59", que representa a nuestra aplicación.

El resultado de ese programa es el siguiente:

Valores iniciales...

Ancho: 0

Alto: 0

Color: 0

Abierta: False

Vamos a abrir...

Ancho: 0

Alto: 0

Color: 0

Abierta: True

Se puede ver que en C#, al contrario que en otros lenguajes, las variables que forman parte de una clase (los "atributos") tienen un valor inicial predefinido: 0 para los números, una cadena vacía para las cadenas de texto, o "False" para los datos booleanos.

Vemos también que se accede a los métodos y a los datos precediendo el nombre de cada uno por el nombre de la variable y por un punto, como hacíamos con los registros (struct). Aun así, en nuestro caso no podemos hacer directamente "p.abierta = true", por dos motivos:

- El atributo "abierta" no tiene delante la palabra "public"; por lo que no es público, sino privado, y no será accesible desde otras clases (en nuestro caso, desde Ejemplo59).
- Los puristas de la Programación Orientada a Objetos recomiendan que no se acceda directamente a los atributos, sino que siempre se modifiquen usando métodos auxiliares (por ejemplo, nuestro "Abrir"), y que se lea su valor también usando una función. Esto es lo que se conoce como "**ocultación de datos**". Supondrá ventajas como que podremos cambiar los detalles internos de nuestra clase sin que afecte a su uso.

Normalmente, como forma de ocultar datos, crearemos funciones auxiliares GetXXX y SetXXX que permitan acceder al valor de los atributos (en C# existe una forma alternativa de hacerlo, usando "propiedades", que veremos más adelante):

```
public int GetAncho()
{
    return ancho;
}

public void SetAncho(int nuevoValor)
{
```

```

    ancho = nuevoValor;
}

```

También puede desconcertar que en "Main" aparezca la palabra "static", mientras que no lo hace en los métodos de la clase "Puerta". Veremos este detalle un poco más adelante.

Ejercicio propuesto:

- Crear una clase llamada Persona, en el fichero "persona.cs". Esta clase deberá tener un atributo "nombre", de tipo string. También deberá tener un método "SetNombre", de tipo void y con un parámetro string, que permita cambiar el valor del nombre. Finalmente, también tendrá un método "Saludar", que escribirá en pantalla "Hola, soy " seguido de su nombre. Crear también una clase llamada PruebaPersona. Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre y les pedirá que saluden.

En un proyecto grande, es recomendable que cada clase esté en su propio fichero fuente, de forma que se puedan localizar con rapidez (en los que hemos hecho en el curso hasta ahora, no era necesario, porque eran muy simples). Precisamente por eso, es interesante (pero no obligatorio) que cada clase esté en un fichero que tenga el mismo nombre: que la clase Puerta se encuentre en el fichero "Puerta.cs". Esta es una regla que no seguiremos en algunos de los ejemplos del texto, por respetar la numeración consecutiva de los ejemplos, pero que sí se debería seguir en un proyecto de mayor tamaño, formado por varias clases.

Para **compilar un programa formado por varios fuentes**, basta con indicar los nombres de todos ellos. Por ejemplo, con Mono sería

```
mcs fuente1.cs fuente2.cs fuente3.cs
```

En ese caso, el ejecutable obtenido tenía el nombre del primero de los fuentes (fuente1.exe). Podemos cambiar el nombre del ejecutable con la opción "-out" de Mono:

```
mcs fuente1.cs fuente2.cs fuente3.cs -out:ejemplo.exe
```

Vamos a dividir en dos fuentes el último ejemplo y a ver cómo se compilaría. La primera clase podría ser ésta:

```

/*-----*/
/* Ejemplo en C# nº 59b:      */
/* ejemplo59b.cs               */
/*                               */
/* Dos clases en dos           */
/* ficheros (fichero 1)        */
/*                               */
/* Introducción a C#,          */
/* Nacho Cabanes               */
/*-----*/

```

```

using System;

public class Puerta
{
    int ancho;      // Ancho en centimetros
    int alto;       // Alto en centimetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
}

} // Final de la clase Puerta

```

Y la segunda clase podría ser:

```

/*
 *-----*
 * Ejemplo en C# nº 59c:      */
 * ejemplo59c.cs               */
 */
/* Dos clases en dos          */
/* ficheros (fichero 2)        */
 */
/* Introducción a C#,          */
/* Nacho Cabanes              */
 *-----*/


public class Ejemplo59c
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales..."); 
        p.MostrarEstado();

        Console.WriteLine("\nVamos a abrir..."); 
        p.Abrir();
        p.MostrarEstado();
    }
}

```

```

    }
}
```

Y lo compilaríamos con:

```
mcs ejemplo59b.cs ejemplo59c.cs -out:ejemplo59byc.exe
```

Aun así, para estos proyectos formados por varias clases, lo ideal es usar algún entorno más avanzado, como SharpDevelop o VisualStudio, que permitan crear todas las clases con comodidad, saltar de una clase a clase otra rápidamente, que marquen dentro del propio editor la línea en la que están los errores... por eso, al final de este tema tendrás un apartado con una introducción al uso de SharpDevelop.

Ejercicio propuesto:

- Modificar el fuente del ejercicio anterior, para dividirlo en dos ficheros: Crear una clase llamada Persona, en el fichero "persona.cs". Esta clase deberá tener un atributo "nombre", de tipo string. También deberá tener un método "SetNombre", de tipo void y con un parámetro string, que permita cambiar el valor del nombre. Finalmente, también tendrá un método "Saludar", que escribirá en pantalla "Hola, soy " seguido de su nombre. Crear también una clase llamada PruebaPersona, en el fichero "pruebaPersona.cs". Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre y les pedirá que saluden.

6.3. La herencia. Visibilidad

Vamos a ver ahora cómo definir una nueva clase de objetos a partir de otra ya existente. Por ejemplo, vamos a crear una clase "Porton" a partir de la clase "Puerta". Un portón tendrá las mismas características que una puerta (ancho, alto, color, abierto o no), pero además se podrá bloquear, lo que supondrá un nuevo atributo y nuevos métodos para bloquear y desbloquear:

```
public class Porton: Puerta
{
    bool bloqueada;

    public void Bloquear()
    {
        bloqueada = true;
    }

    public void Desbloquear()
    {
        bloqueada = false;
    }
}
```

Con "public class Porton: Puerta" indicamos que Porton debe "heredar" todo lo que ya habíamos definido para Puerta. Por eso, no hace falta indicar nuevamente que un Portón tendrá un cierto ancho, o un color, o que se puede abrir: todo eso lo tiene por ser un "descendiente" de Puerta.

No tenemos por qué heredar todo; también podemos "redefinir" algo que ya existía. Por ejemplo, nos puede interesar que "MostrarEstado" ahora nos diga también si la puerta está bloqueada. Para eso, basta con volverlo a declarar y añadir la palabra "**new**" para indicar al compilador de C# que sabemos que ya existe ese método y que sabemos seguro que lo queremos redefinir:

```
public new void MostrarEstado()
{
    Console.WriteLine("Ancho: {0}", ancho);
    Console.WriteLine("Alto: {0}", alto);
    Console.WriteLine("Color: {0}", color);
    Console.WriteLine("Abierta: {0}", abierta);
    Console.WriteLine("Bloqueada: {0}", bloqueada);
}
```

Aun así, esto todavía no funciona: los atributos de una Puerta, como el "ancho" y el "alto" estaban declarados como "privados" (es lo que se considera si no decimos lo contrario), por lo que no son accesibles desde ninguna otra clase, ni siquiera desde Porton.

La solución más razonable no es declararlos como "public", porque no queremos que sean accesibles desde cualquier sitio. Sólo queríamos que esos datos estuvieran disponibles para todos los tipos de Puerta, incluyendo sus "descendientes", como un Porton. Esto se puede conseguir usando otro método de acceso: "**protected**". Todo lo que declaremos como "protected" será accesible por las clases derivadas de la actual, pero por nadie más:

```
public class Puerta
{
    protected int ancho;      // Ancho en centímetros
    protected int alto;       // Alto en centímetros
    protected int color;      // Color en formato RGB
    protected bool abierta;   // Abierta o cerrada

    public void Abrir()
    ...
}
```

(Si quisiéramos dejar claro que algún elemento de una clase debe ser totalmente privado, podemos usar la palabra "**private**", en vez de "public" o "protected").

Un fuente completo que declarase la clase Puerta, la clase Porton a partir de ella, y que además contuviese un pequeño "Main" de prueba podría ser:

```
/*-----*/
/* Ejemplo en C# nº 60: */
/* ejemplo60.cs */
/*-----*/
```

```

/*
 * Segundo ejemplo de
 * clases: herencia
 *
 * Introducción a C#,
 * Nacho Cabanes
 */
-----*/
using System;

// -----
public class Puerta
{
    protected int ancho;      // Ancho en centímetros
    protected int alto;        // Alto en centímetros
    protected int color;       // Color en formato RGB
    protected bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
}

} // Final de la clase Puerta

// -----
public class Porton: Puerta
{
    bool bloqueada;

    public void Bloquear()
    {
        bloqueada = true;
    }

    public void Desbloquear()
    {
        bloqueada = false;
    }

    public new void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
    }
}

```

```

        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
        Console.WriteLine("Bloqueada: {0}", bloqueada);
    }

} // Final de la clase Porton

// -----
public class Ejemplo60
{
    public static void Main()
    {
        Porton p = new Porton();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine("\nVamos a bloquear...");
        p.Bloquear();
        p.MostrarEstado();

        Console.WriteLine("\nVamos a desbloquear y a abrir...");
        p.Desbloquear();
        p.Abrir();
        p.MostrarEstado();
    }
}

```

Ejercicios propuestos:

- Ampliar las clases del primer ejercicio propuesto, creando un nuevo proyecto con las siguientes características: La clase Persona no cambia. Se creará una nueva clase PersonaIngresa, en el fichero "personaIngresa.cs". Esta clase deberá heredar las características de la clase "Persona", y añadir un método "TomarTe", de tipo void, que escribirá en pantalla "Estoy tomando té". Crear también una clase llamada PruebaPersona2, en el fichero "pruebaPersona2.cs". Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona y uno de tipo PersonaIngresa, les asignará un nombre, les pedirá que saluden y pedirá a la persona ingresa que tome té.
- Ampliar las clases del segundo ejercicio propuesto, creando un nuevo proyecto con las siguientes características: La clase Persona no cambia. La clase PersonaIngresa se modificará para que redefina el método "Saludar", para que escriba en pantalla "Hi, I am " seguido de su nombre. Se creará una nueva clase PersonaItaliana, en el fichero "personaItaliana.cs". Esta clase deberá heredar las características de la clase "Persona", pero redefinir el método "Saludar", para que escriba en pantalla "Ciao". Crear también una clase llamada PruebaPersona3, en el fichero "pruebaPersona3.cs". Esta clase deberá contener sólo la función Main, que creará un objeto de tipo Persona, dos de tipo PersonaIngresa, uno de tipo PersonaItaliana, les asignará un nombre, les pedirá que saluden y pedirá a la persona ingresa que tome té.

6.4. ¿Cómo se diseñan las clases?

En estos primeros ejemplos, hemos "pensado" qué objetos necesitaríamos, y hemos empezado a teclear directamente para implementarlos. Esto no es lo habitual. Normalmente, se usan **herramientas gráficas** que nos ayuden a visualizar las clases y las relaciones que existen entre ellas. También se puede dibujar directamente en papel para aclararnos las ideas, pero el empleo de herramientas informáticas tiene una ventaja adicional: algunas de ellas nos permiten generar automáticamente un esqueleto del programa.

La metodología más extendida actualmente para diseñar estos objetos y sus interacciones (además de otras muchas cosas) se conoce como **UML** (Unified Modelling Language, lenguaje de modelado unificado). El estándar UML propone distintos tipos de diagramas para representar los posibles "casos de uso" de una aplicación, la secuencia de acciones que se debe seguir, las clases que la van a integrar (es lo que a nosotros nos interesa en este momento), etc.

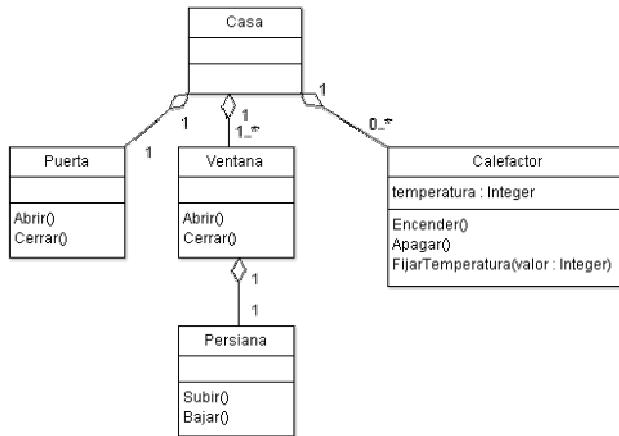
Vamos a ver la apariencia que tendría un "**diagrama de clases**". En concreto, vamos a ver un ejemplo usando **ArgoUML**, que es una herramienta gratuita de modelado UML, que está creada en Java, por lo que se puede utilizar desde multitud de sistemas operativos.

Ampliando el ejemplo anterior, vamos a suponer que queremos hacer un sistema "domótico", para automatizar ciertas funciones en una casa: apertura y cierre de ventanas y puertas, encendido de calefacción, etc.

Las ideas iniciales de las que partiremos son:

- La casa es el conjunto ("agregación") de varios elementos: puertas, ventanas y calefactores.
- Cada puerta se puede abrir y cerrar.
- Cada ventana se puede abrir, cerrar. Además, las ventanas tienen persianas, que se pueden subir y bajar.
- Cada calefactor puede encenderse, apagarse o se puede programar para que trabaje a una cierta temperatura.

Con estas posibilidades básicas, el diagrama de clases podría ser así:



Este diagrama es una forma más simple de ver las clases existentes y las relaciones entre ellas. Si generamos las clases a partir del diagrama, tendremos parte del trabajo hecho: ya "sólo" nos quedará llenar los detalles de métodos como "Abrir", pero el esqueleto de todas las clases ya estará "escrito" para nosotros.

6.5. La palabra "static"

Desde un principio, nos hemos encontrado con que "Main" siempre iba acompañado de la palabra "static". En cambio, los métodos (funciones) que pertenecen a nuestros objetos no los estamos declarando como "static". Vamos a ver el motivo:

La palabra "static" delante de un atributo (una variable) de una clase, indica que es una "variable de clase", es decir, que su valor es el mismo para todos los objetos de la clase. Por ejemplo, si hablamos de coches convencionales, podríamos suponer que el atributo "numeroDeRuedas" va a valer 4 para cualquier objeto que pertenezca a esa clase (cualquier coches). Por eso, se podría declarar como "static".

De igual modo, si un método (una función) está precedido por la palabra "static", indica que es un "método de clase", es decir, un método que se podría usar sin necesidad de declarar ningún objeto de la clase. Por ejemplo, si queremos que se pueda usar la función "BorrarPantalla" de una clase "Hardware" sin necesidad de crear primero un objeto perteneciente a esa clase, lo podríamos conseguir así:

```

public class Hardware
{
    ...
    public static void BorrarPantalla ()
    {
        ...
    }
}
  
```

que desde dentro de "Main" (incluso perteneciente a otra clase) se usaría con el nombre de la clase delante:

```
public class Juego
{
    ...
    public ComienzoPartida()
    {
        Hardware.BorrarPantalla();
        ...
    }
}
```

Desde una función "static" no se puede llamar a otras funciones que no lo sean. Por eso, como nuestro "Main" debe ser static, deberemos siempre elegir entre:

- Que todas las demás funciones de nuestro fuente también estén declaradas como "static", por lo que podrán ser utilizadas desde "Main".
- Declarar un objeto de la clase correspondiente, y entonces sí podremos acceder a sus métodos desde "Main":

```
public class Ejemplo
{
    ...
    public LanzarJuego()
    {
        Juego j = new Juego();
        j.ComienzoPartida();
        ...
    }
}
```

6.6. Constructores y destructores.

Hemos visto que al declarar una clase, se dan valores por defecto para los atributos. Por ejemplo, para un número entero, se le da el valor 0. Pero puede ocurrir que nosotros deseemos dar valores iniciales que no sean cero. Esto se puede conseguir declarando un "**constructor**" para la clase.

Un constructor es una función especial, que se pone en marcha cuando se crea un objeto de una clase, y se suele usar para dar esos valores iniciales, para reservar memoria si fuera necesario, etc.

Se declara usando el mismo nombre que el de la clase, y sin ningún tipo de retorno. Por ejemplo, un "constructor" para la clase Puerta que le diera los valores iniciales de 100 para el ancho, 200 para el alto, etc., podría ser así:

```
public Puerta()
{
    ancho = 100;
    alto = 200;
    color = 0xFFFFFFFF;
    abierta = false;
```

```

}
```

Podemos tener más de un constructor, cada uno con distintos parámetros. Por ejemplo, puede haber otro constructor que nos permita indicar el ancho y el alto:

```

public Puerta(int an, int al)
{
    ancho = an;
    alto = al;
    color = 0xFFFFFFFF;
    abierta = false;
}
```

Ahora, si declaramos un objeto de la clase puerta con "Puerta p = new Puerta();" tendrá de ancho 100 y de alto 200, mientras que si lo declaramos con "Puerta p2 = new Puerta(90,220);" tendrá 90 como ancho y 220 como alto.

Un programa de ejemplo que usara estos dos constructores para crear dos puertas con características iniciales distintas podría ser:

```

/*-----*/
/* Ejemplo en C# nº 61:      */
/* ejemplo61.cs               */
/*                               */
/* Tercer ejemplo de clases   */
/* Constructores               */
/*                               */
/* Introduccion a C#,          */
/* Nacho Cabanes               */
/*-----*/
using System;

public class Puerta
{
    int ancho;      // Ancho en centimetros
    int alto;       // Alto en centimetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public Puerta()
    {
        ancho = 100;
        alto = 200;
        color = 0xFFFFFFFF;
        abierta = false;
    }

    public Puerta(int an, int al)
    {
        ancho = an;
        alto = al;
        color = 0xFFFFFFFF;
```

```

        abierta = false;
    }

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
}

// Final de la clase Puerta

public class Ejemplo61
{
    public static void Main()
    {
        Puerta p = new Puerta();
        Puerta p2 = new Puerta(90,220);

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine("\nVamos a abrir...");
        p.Abrir();
        p.MostrarEstado();

        Console.WriteLine("Para la segunda puerta...");
        p2.MostrarEstado();
    }
}

```

Nota: al igual que existen los "constructores", también podemos indicar un "**destructor**" para una clase, que se encargue de liberar la memoria que pudiéramos haber reservado en nuestra clase (no es nuestro caso, porque aún no sabemos manejar memoria dinámica) o para cerrar ficheros abiertos (que tampoco sabemos).

Un "destructor" se llama igual que la clase, pero precedido por el símbolo "~", no tiene tipo de retorno, y no necesita ser "public", como ocurre en este ejemplo:

```

~Puerta()
{
    // Liberar memoria
}

```

```
// Cerrar ficheros
}
```

6.7. Polimorfismo y sobrecarga

Esos dos constructores "Puerta()" y "Puerta(int ancho, int alto)", que se llaman igual pero reciben distintos parámetros, y se comportan de forma que puede ser distinta, son ejemplos de "**polimorfismo**" (funciones que tienen el mismo nombre, pero distintos parámetros, y que quizás no se comporten de igual forma).

Un concepto muy relacionado con el polimorfismo es el de "**sobrecarga**": dos funciones están sobrecargadas cuando se llaman igual, reciben el mismo número de parámetros, pero se aplican a objetos distintos, así:

```
puerta.Abrir ();
libro.Abrir ();
```

En este caso, la función "Abrir" está sobrecargada: se usa tanto para referirnos a abrir un libro como para abrir una puerta. Se trata de dos acciones que no son exactamente iguales, que se aplican a objetos distintos, pero que se llaman igual.

6.8. Orden de llamada de los constructores

Cuando creamos objetos de una clase derivada, antes de llamar a su constructor se llama a los constructores de las clases base, empezando por la más general y terminando por la más específica. Por ejemplo, si creamos una clase "GatoSiamés", que deriva de una clase "Gato", que a su vez procede de una clase "Animal", el orden de ejecución de los constructores sería: Animal, Gato, GatoSiames, como se ve en este ejemplo:

```
/*-----*/
/* Ejemplo en C# nº 62:      */
/* ejemplo62.cs               */
/*                           */
/* Cuarto ejemplo de clases */
/* Constructores y herencia */
/*                           */
/* Introducción a C#,          */
/* Nacho Cabanes              */
/*-----*/
```

```
using System;

public class Animal
{
    public Animal()
    {
```

```

        Console.WriteLine("Ha nacido un animal");
    }

}

// ----

public class Perro: Animal
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}

// ----

public class Gato: Animal
{
    public Gato()
    {
        Console.WriteLine("Ha nacido un gato");
    }
}

// ----

public class GatoSiames: Gato
{
    public GatoSiames()
    {
        Console.WriteLine("Ha nacido un gato siamés");
    }
}

// ----

public class Ejemplo62
{
    public static void Main()
    {
        Animal a1      = new Animal();
        GatoSiames a2 = new GatoSiames();
        Perro a3       = new Perro();
        Gato a4        = new Gato();
    }
}

```

El resultado de este programa es:

```

Ha nacido un animal
Ha nacido un animal

```

```

Ha nacido un gato
Ha nacido un gato siamés
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un gato

```

Ejercicio propuesto:

- Crear un único fuente que contenga las siguientes clases:
 - Una clase Trabajador, cuyo constructor escriba en pantalla "Soy un trabajador".
 - Una clase Programador, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy programador".
 - Una clase Analista, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy analista".
 - Una clase Ingeniero, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy ingeniero".
 - Una clase IngenieroInformatico, que derive de Ingeniero, cuyo constructor escriba en pantalla "Soy ingeniero informático".
 - Una clase "PruebaDeTrabajadores", que cree un objeto perteneciente a cada una de esas clases.

6.9. Arrays de objetos

Es muy frecuente que no nos baste con tener un objeto de cada clase, sino que necesitemos manipular varios objetos pertenecientes a la misma clase.

En ese caso, deberemos reservar memoria primero para el array, y luego para cada uno de los elementos. Por ejemplo, podríamos tener un array de 5 perros, que crearíamos de esta forma:

```

Perro[] misPerros = new Perro[5];
for (byte i = 0; i < 5; i++)
    misPerros[i] = new Perro();

```

Un fuente completo de ejemplo podría ser

```

/*-----*/
/* Ejemplo en C# nº 63:      */
/* ejemplo63.cs               */
/*                           */
/* Quinto ejemplo de clases */
/* Array de objetos           */
/*                           */
/* Introduccion a C#,        */
/* Nacho Cabanes             */
/*-----*/

```

```

using System;

public class Animal

```

```

{
    public Animal()
    {
        Console.WriteLine("Ha nacido un animal");
    }
}

// ----

public class Perro: Animal
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}

// ----

public class Ejemplo63
{
    public static void Main()
    {
        Perro[] misPerros = new Perro[5];
        for (byte i = 0; i < 5; i++)
            misPerros[i] = new Perro();
    }
}

```

y su salida en pantalla, parecida a la del ejemplo anterior, sería

```

Ha nacido un animal
Ha nacido un perro

```

Ejercicio propuesto:

- Crea una versión ampliada del anterior ejercicio propuesto, en la que no se cree un único objeto de cada clase, sino un array de tres objetos.

Además, existe una peculiaridad curiosa: podemos crear un array de "Animales", pero luego indicar que unos de ellos son perros, otros gatos, etc.,

```
Animal[] misAnimales = new Animal[3];

misAnimales[0] = new Perro();
misAnimales[1] = new Gato();
misAnimales[2] = new GatoSiames();
```

Un ejemplo más detallado:

```
/*
 * Ejemplo en C# nº 64:
 * ejemplo64.cs
 *
 * Ejemplo de clases
 * Array de objetos de
 * varias subclases
 *
 * Introducción a C#
 * Nacho Cabanes
 */

using System;

public class Animal
{
    public Animal()
    {
        Console.WriteLine("Ha nacido un animal");
    }
}

// ----

public class Perro: Animal
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}

// ----

public class Gato: Animal
{
    public Gato()
    {
        Console.WriteLine("Ha nacido un gato");
    }
}

// -----
```

```
public class GatoSiames: Gato
{
    public GatoSiames()
    {
        Console.WriteLine("Ha nacido un gato siamés");
    }
}

// -----

public class Ejemplo64
{
    public static void Main()
    {
        Animal[] misAnimales = new Animal[8];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new GatoSiames();

        for (byte i=3; i<7; i++)
            misAnimales[i] = new Perro();

        misAnimales[7] = new Animal();
    }
}
```

La salida de este programa sería:

```
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un gato
Ha nacido un animal
Ha nacido un gato
Ha nacido un gato siamés
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
```

6.10. Funciones virtuales. La palabra "override"

En el ejemplo anterior hemos visto cómo crear un array de objetos, usando sólo la clase base, pero insertando realmente objetos de cada una de las clases derivadas que nos interesaba, y hemos visto que los constructores se llaman correctamente... pero con los métodos puede haber problemas.

Vamos a verlo con un ejemplo, que en vez de tener constructores va a tener un único método "Hablar", que se redefine en cada una de las clases hijas, y después comentaremos qué ocurre al ejecutarlo:

```
/*-----*/
/* Ejemplo en C# nº 65: */
/* ejemplo65.cs */
/*
/*
/* Ejemplo de clases */
/* Array de objetos de */
/* varias subclases con */
/* metodos */
/*
/*
/* Introduccion a C#,
/* Nacho Cabanes */
/*
// -----



using System;

public class Animal
{
    public void Hablar()
    {
        Console.WriteLine("Estoy comunicándome...");
    }
}

// -----


public class Perro: Animal
{
    public new void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}

// -----


public class Gato: Animal
{
    public new void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}
```

```
// ----

public class Ejemplo65
{
    public static void Main()
    {
        // Primero creamos un animal de cada tipo
        Perro miPerro = new Perro();
        Gato miGato = new Gato();
        Animal miAnimal = new Animal();

        miPerro.Hablar();
        miGato.Hablar();
        miAnimal.Hablar();

        // Línea en blanco, por legibilidad
        Console.WriteLine();

        // Ahora los creamos desde un array
        Animal[] misAnimales = new Animal[3];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new Animal();

        misAnimales[0].Hablar();
        misAnimales[1].Hablar();
        misAnimales[2].Hablar();
    }
}
```

La salida de este programa es:

```
Guau!
Miauuu
Estoy comunicándome...

Estoy comunicándome...
Estoy comunicándome...
Estoy comunicándome...
```

La primera parte era de esperar: si creamos un perro, debería decir "Guau", un gato debería decir "Miau" y un animal genérico debería comunicarse. Eso es lo que se consigue con este fragmento:

```
Perro miPerro = new Perro();
Gato miGato = new Gato();
Animal miAnimal = new Animal();

miPerro.Hablar();
miGato.Hablar();
miAnimal.Hablar();
```

En cambio, si creamos un array de animales, no se comporta correctamente, a pesar de que después digamos que el primer elemento del array es un perro:

```
Animal[] misAnimales = new Animal[3];

misAnimales[0] = new Perro();
misAnimales[1] = new Gato();
misAnimales[2] = new Animal();

misAnimales[0].Hablar();
misAnimales[1].Hablar();
misAnimales[2].Hablar();
```

Es decir, como la clase base es "Animal", el primer elemento hace lo que corresponde a un Animal genérico (decir "Estoy comunicándome"), a pesar de que hayamos dicho que se trata de un Perro.

Generalmente, no será esto lo que queramos. Sería interesante no necesitar crear un array de perros y otros de gatos, sino poder crear un array de animales, y que contuviera animales de distintos tipos.

Para conseguir este comportamiento, debemos indicar a nuestro compilador que el método "Hablar" que se usa en la clase Animal puede que sea redefinido por otras clases hijas, y que en ese caso debe prevalecer lo que indiquen las clases hijas.

La forma de hacerlo es declarando ese método "Hablar" como "virtual", y empleando en las clases hijas la palabra "override" en vez de "new", así:

```
/*
 * Ejemplo en C# nº 66:
 */
/* ejemplo66.cs
 */
/* Ejemplo de clases
 */
/* Array de objetos de
 */
/* varias subclases con
 */
/* metodos virtuales
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes
 */
using System;

public class Animal
{
    public virtual void Hablar()
    {
        Console.WriteLine("Estoy comunicándome... ");
    }
}
```

```

// ----

public class Perro: Animal
{
    public override void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}

// ----

public class Gato: Animal
{
    public override void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}

// ----

public class Ejemplo66
{
    public static void Main()
    {
        // Primero creamos un animal de cada tipo
        Perro miPerro = new Perro();
        Gato miGato = new Gato();
        Animal miAnimal = new Animal();

        miPerro.Hablar();
        miGato.Hablar();
        miAnimal.Hablar();

        // Linea en blanco, por legibilidad
        Console.WriteLine();

        // Ahora los creamos desde un array
        Animal[] misAnimales = new Animal[3];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new Animal();

        misAnimales[0].Hablar();
        misAnimales[1].Hablar();
        misAnimales[2].Hablar();
    }
}

```

El resultado de este programa ya sí es el que posiblemente deseábamos: tenemos un array de animales, pero cada uno "Habla" como corresponde a su especie:

Guau!
 Miauuu
 Estoy comunicándome...

Guau!
 Miauuu
 Estoy comunicándome...

6.11. Llamando a un método de la clase "padre"

Puede ocurrir que en un método de una clase hija no nos interese redefinir por completo las posibilidades del método equivalente, sino ampliarlas. En ese caso, no hace falta que volvamos a teclear todo lo que hacía el método de la clase base, sino que podemos llamarlo directamente, precediéndolo de la palabra "base". Por ejemplo, podemos hacer que un Gato Siamés hable igual que un Gato normal, pero diciendo "Pfff" después, así:

```
public new void Hablar()
{
    base.Hablar();
    Console.WriteLine("Pfff");
}
```

Este podría ser un fuente completo:

```
/*-----*/
/* Ejemplo en C# nº 67:      */
/* ejemplo67.cs               */
/*-----*/
/* Ejemplo de clases          */
/* Llamar a la superclase     */
/*-----*/
/* Introducción a C#,          */
/* Nacho Cabanes              */
/*-----*/
```

```
using System;

public class Animal
{
}

// -----

public class Gato : Animal
{
    public void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}
```

```
// ----

public class GatoSiames: Gato
{
    public new void Hablar()
    {
        base.Hablar();
        Console.WriteLine("Pfff");
    }
}

// ----

public class Ejemplo67
{
    public static void Main()
    {
        Gato miGato = new Gato();
        GatoSiames miGato2 = new GatoSiames();

        miGato.Hablar();
        Console.WriteLine(); // Linea en blanco
        miGato2.Hablar();
    }
}
```

Su resultado sería

Miauuu

Miauuu

Pfff

También podemos llamar a un **constructor** de la clase base desde un constructor de una clase derivada. Por ejemplo, si tenemos una clase "RectanguloRelleno" que hereda de otra clase "Rectangulo" y queremos que el constructor de "RectanguloRelleno" que recibe las coordenadas "x" e "y" se base en el constructor equivalente de la clase "Rectangulo", lo haríamos así:

```
public RectanguloRelleno (int x, int y )
    : base (x, y)
{
    // Pasos adicionales
    // que no da un rectangulo "normal"
}
```

(Si no hacemos esto, el constructor de RectanguloRelleno se basaría en el constructor sin parámetros de Rectangulo, no en el que tiene x e y como parámetros).

6.12. La palabra "this": el objeto actual

Podemos hacer referencia al objeto que estamos usando, con "this":

```
public void MoverA (int x, int y)
{
    this.x = x;
    this.y = y;
}
```

En muchos casos, podemos evitarlo. Por ejemplo, normalmente es preferible usar otro nombre en los parámetros:

```
public void MoverA (int nuevaX, int nuevaY)
{
    this.x = nuevaX;
    this.y = nuevaY;
}
```

Y en ese uso se puede (y se suele) omitir "this":

```
public void MoverA (int nuevaX, int nuevaY)
{
    x = nuevaX;
    y = nuevaY;
}
```

Pero "this" tiene también otros usos. Por ejemplo, podemos crear un **constructor** a partir de otro que tenga distintos parámetros:

```
public RectanguloRelleno (int x, int y )
    : this (x)
{
    fila = y;
    // Pasos adicionales
}
```

Y se usa mucho para que unos objetos "conozcan" a los otros:

```
public void IndicarEnemigo (ElemGrafico enemigo)
{
    ...
}
```

De modo que el personaje de un juego le podría indicar al adversario que él es su enemigo con

```
miAdversario.IndicarEnemigo(this);
```

Es habitual usarlo para que una clase sepa a cual pertenece (por ejemplo, a qué casa pertenece una puerta), porque de lo contrario, cada clase "conoce" a los objetos que contiene (la casa "conoce" a sus puertas), pero no al contrario.

6.13. Sobrecarga de operadores

Los "operadores" son los símbolos que se emplean para indicar ciertas operaciones. Por ejemplo, el operador "+" se usa para indicar que queremos sumar dos números.

Pues bien, en C# se puede "sobrecargar" operadores, es decir, redefinir su significado, para poder sumar (por ejemplo) objetos que nosotros hayamos creado, de forma más cómoda y legible. Por ejemplo, para sumar dos matrices, en vez de hacer algo como "matriz3 = suma(matriz1, matriz2)" podríamos hacer simplemente " matriz3 = matriz1 + matriz2"

No entraremos en detalle, pero la idea está en que redefiniríamos un método llamado "operador +", y que devolvería un dato del tipo con el que estamos (por ejemplo, una Matriz) y recibiría dos datos de ese mismo tipo como parámetros:

```
public static Matriz operator +(Matriz mat1, Matriz mat2)
{
    Matriz nuevaMatriz = new Matriz();

    for (int x=0; x < tamano; x++)
        for (int y=0; y < tamano; y++)
            nuevaMatriz[x, y] = mat1[x, y] + mat2[x, y];

    return nuevaMatriz;
}
```

Desde "Main", calcularíamos una matriz como suma de otras dos haciendo simplemente

```
Matriz matriz3 = matriz1 + matriz2;
```

Ejercicios propuestos:

- Desarrolla una clase "Matriz", que represente a una matriz de 3x3, con métodos para indicar el valor que hay en una posición, leer el valor de una posición, escribir la matriz en pantalla y sumar dos matrices.

6.14. Proyectos a partir de varios fuentes

En los programas de gran tamaño, lo habitual es que no se plantee como un único fichero fuente de gran tamaño, sino como una serie de objetos que colaboran entre ellos, lo que supone descomponer nuestro fuente en varias clases (y, por tanto, en varios ficheros).

En estos casos, un editor como el Bloc de Notas, o incluso como Notepad++, se queda corto. Sería preferible un entorno que nos permitiera editar nuestros fuentes, compilarlos sin necesidad de salir de él, que nos destacara las líneas que contienen errores, que nos mostrara ayuda sobre la sintaxis de cada función, que nos permitiera depurar nuestros programas avanzando paso a paso...

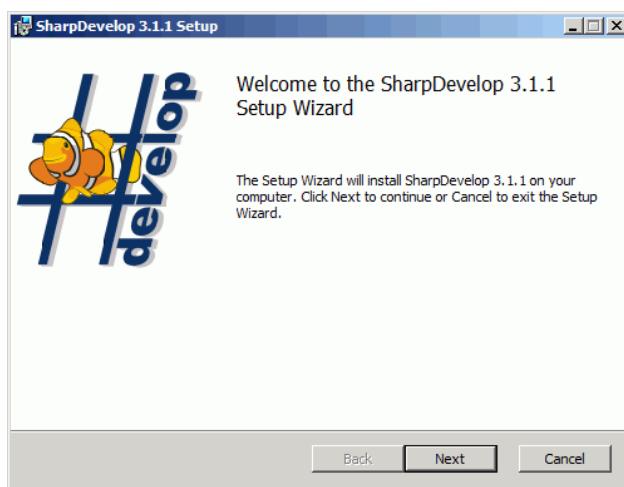
Existen entornos que nos permiten hacer todo eso, y además hacerlo gratis. El más conocido es el Visual Studio de Microsoft, que en su versión Express incluye todo lo un programador novel

como nosotros puede necesitar. Una alternativa muy similar, pero algo más sencilla (lo que supone que funcione más rápido en ordenadores no demasiado potentes) es SharpDevelop.

Vamos a ver las pautas básicas de manejo de **SharpDevelop** (que se aplicarían con muy pocos cambios al caso de Visual Studio):

Comenzamos por descargar el fichero de instalación del entorno, desde su página oficial (<http://www.icsharpcode.netopensource/sd/>). La versión 3.1, para las versiones 2.0 y 3.5 de la plataforma .Net, ocupa unos 15 Mb.

La instalación comenzará simplemente con hacer doble clic. Deberíamos ver una ventana parecida a ésta:

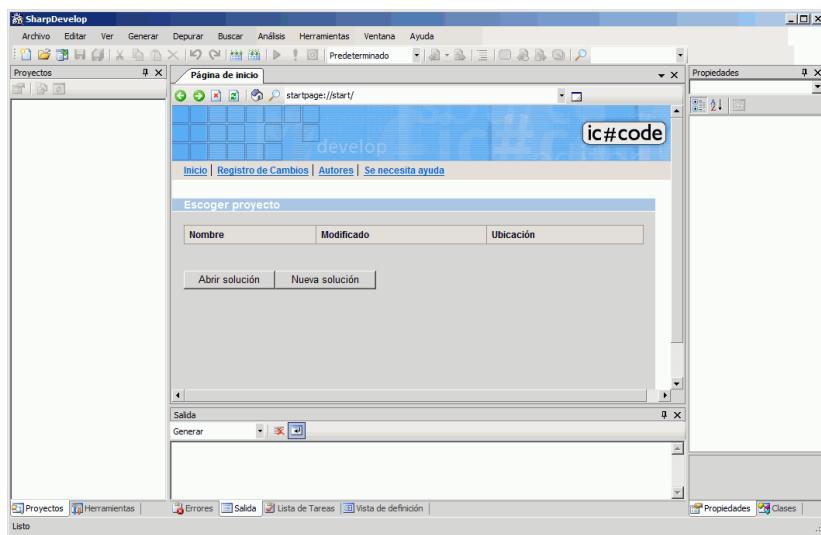


Como es habitual, el siguiente paso será aceptar el contrato de licencia, después deberemos decir en qué carpeta queremos instalarlo, comenzará a copiar archivos y al cabo de un instante, tendremos un nuevo ícono en nuestro escritorio:

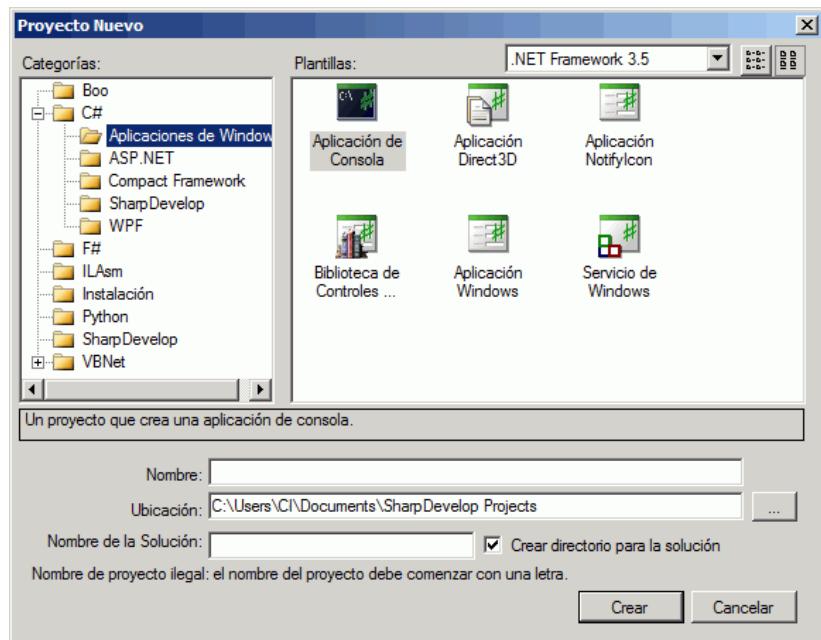


La instalación debería ser muy sencilla en Windows Vista y superiores, pero en Windows XP quizás necesite que instalamos la versión 3.5 de la plataforma .Net (se puede hacer gratuitamente desde su página oficial).

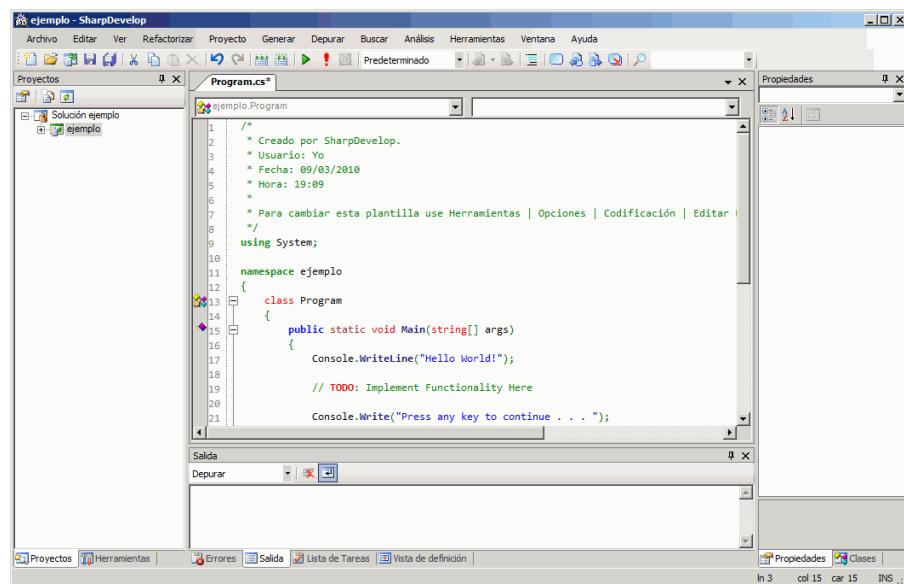
Cuando lanzamos nuestro nuevo ícono, veremos la pantalla principal de SharpDevelop, que nos muestra la lista de los últimos proyectos ("soluciones") que hemos realizado, y nos permite crear uno nuevo:



En nuestro caso, comenzaremos por crear una "Nueva solución", y se nos mostrará los tipos de proyectos para los que se nos podría crear un esqueleto vacío que después iríamos rellenando:



De estos tipos, el único que conocemos es una "Aplicación de Consola" (en C#, claro). Deberemos escribir también el nombre., y aparecerá un esqueleto de aplicación que nosotros sólo tendríamos que completar:



Cuando hayamos terminado de realizar nuestros cambios, podemos compilar el programa con el botón:

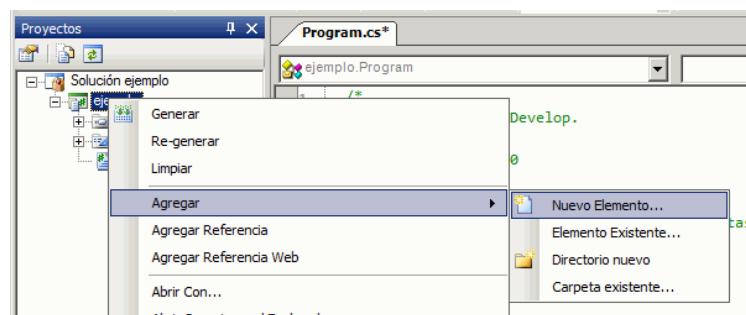


Si hubiera algún error, se nos avisaría en la parte inferior de la pantalla, y se subrayarían en rojo las líneas correspondientes de nuestro programa; si todo ha ido bien, podremos ejecutar nuestro programa para verlo funcionando:

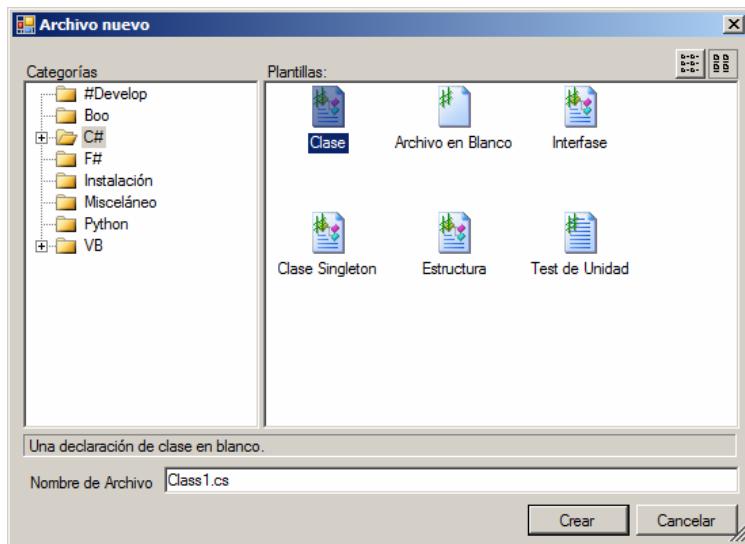


(Si la ventana de nuestro programa se cierra tan rápido que no tenemos tiempo de leerla, nos puede interesar añadir provisionalmente una línea ReadLine() al final del fuente, para que éste se detenga hasta que pulsemos la tecla Intro)

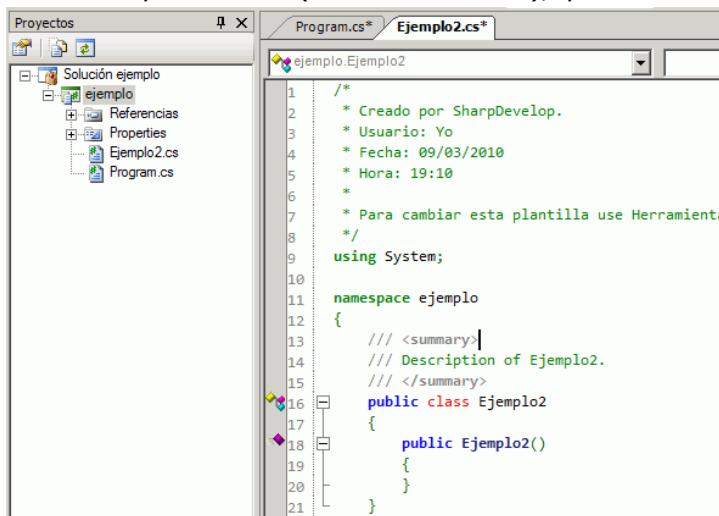
Así prepararíamos y lanzaríamos un programa formado por un solo fuente. Si se trata de varios fuentes, basta con ir añadiendo nuevas clases al proyecto. Lo conseguimos pulsando el botón derecho sobre el nombre del proyecto (en la ventana izquierda, "Proyectos") y escogiendo las opciones Agregar / Nuevo Elemento:



Normalmente, el tipo de elemento que nos interesará será una clase, cuyo nombre deberemos indicar:



y obtendríamos un nuevo esqueleto vacío (esta vez sin "Main"), que deberíamos completar.



Nuestro programa, que ahora estaría formado por dos clases, se compilaría y se ejecutaría de la misma forma que cuando estaba integrado por una única clase.

Ejercicio propuesto:

- Crear un proyecto que contenga las siguientes clases (cada una en un fichero distinto):
 - Una clase Trabajador, cuyo constructor escriba en pantalla "Soy un trabajador".
 - Una clase Programador, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy programador".
 - Una clase Analista, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy analista".
 - Una clase Ingeniero, que derive de Trabajador, cuyo constructor escriba en pantalla "Soy ingeniero".

- Una clase IngenieroInformatico, que derive de Ingeniero, cuyo constructor escriba en pantalla "Soy ingeniero informático".
- Una clase "PruebaDeTrabajadores", que cree un objeto perteneciente a cada una de esas clases.

7. Manejo de ficheros

7.1. Escritura en un fichero de texto

Para manejar ficheros, siempre deberemos realizar tres operaciones básicas:

- Abrir el fichero.
- Leer datos de él o escribir datos en él.
- Cerrar el fichero.

Eso sí, no siempre podremos realizar esas operaciones, así que además tendremos que comprobar los posibles errores. Por ejemplo, puede ocurrir que intentemos abrir un fichero que realmente no exista, o que queramos escribir en un dispositivo que sea sólo de lectura.

Vamos a ver un ejemplo, que cree un fichero de texto y escriba algo en él:

```
/*
 *----- Ejemplo en C# nº 70 -----
 *----- ejemplo70.cs -----
 */
/*----- Escritura en un fichero de texto -----*/
/*----- Introducción a C#, Nacho Cabanes -----*/
/*----- */

using System;
using System.IO; // Para StreamWriter

public class Ejemplo70
{
    public static void Main()
    {
        StreamWriter fichero;

        fichero = File.CreateText("prueba.txt");
        fichero.WriteLine("Esto es una línea");
        fichero.Write("Esto es otra");
        fichero.WriteLine(" y esto es continuación de la anterior");
        fichero.Close();
    }
}
```

Hay varias cosas que comentar sobre este programa:

- StreamWriter es la clase que representa a un fichero en el que podemos escribir.
- El fichero de texto lo creamos con el método CreateText, que pertenece a la clase File.
- Para escribir en el fichero, empleamos Write y WriteLine, al igual que en la consola.

- Finalmente, debemos cerrar el fichero con Close, o podría quedar algún dato sin guardar.

Ejercicios propuestos:

- Crea un programa que vaya leyendo las frases que el usuario teclea y las guarde en un fichero de texto llamado "registroDeUsuario.txt". Terminará cuando la frase introducida sea "fin" (esa frase no deberá guardarse en el fichero).

7.2. Lectura de un fichero de texto

La estructura de un programa que leyera de un fichero de texto sería parecida:

```
/*
 *----- Ejemplo en C# nº 71:
 *----- ejemplo71.cs
 */
/*
 *----- Lectura de un fichero de
 *----- texto
 */
/*
 *----- Introducción a C#,
 *----- Nacho Cabanes
 */
using System;
using System.IO; // Para StreamReader

public class Ejemplo71
{
    public static void Main()
    {
        StreamReader fichero;
        string linea;

        fichero = File.OpenText("prueba.txt");
        linea = fichero.ReadLine();
        Console.WriteLine( linea );
        Console.WriteLine( fichero.ReadLine() );
        fichero.Close();
    }
}
```

Las diferencias son:

- Para leer de un fichero no usaremos StreamWriter, sino StreamReader.
- Si queremos abrir un fichero que ya existe, usaremos OpenText, en lugar de CreateText.
- Para leer del fichero, usaríamos ReadLine, como hacíamos en la consola.
- Nuevamente, deberemos cerrar el fichero al terminar de usarlo.

Ejercicios propuestos:

- Crea un programa que lea las tres primeras líneas del fichero creado en el apartado anterior y las muestre en pantalla.

7.3. Lectura hasta el final del fichero

Normalmente no querremos leer sólo una frase del fichero, sino procesar todo su contenido, de principio a fin

En un fichero de texto, la forma de saber si hemos llegado al final es intentar leer una línea, y comprobar si el resultado ha sido "null", lo que nos indicaría que no se ha podido leer y que, por tanto estamos en el final del fichero.

Normalmente, si queremos procesar todo un fichero, esta lectura y comprobación debería ser repetitiva, así:

```
/*-----*/
/* Ejemplo en C# nº 72:      */
/* ejemplo72.cs              */
/*                           */
/* Lectura de un fichero de */
/* texto                     */
/*                           */
/* Introduccion a C#,        */
/* Nacho Cabanes             */
/*-----*/
using System;
using System.IO; // Para StreamReader

public class Ejemplo72
{
    public static void Main()
    {
        StreamReader fichero;
        string linea;

        fichero = File.OpenText("prueba.txt");
        do
        {
            linea = fichero.ReadLine();
            if (linea != null)
                Console.WriteLine( linea );
        }
        while (linea != null);

        fichero.Close();
    }
}
```

Ejercicios propuestos:

- Un programa que pida al usuario que teclee frases, y las almacene en el fichero "frases.txt". Acabará cuando el usuario pulse Intro sin teclear nada. Después deberá mostrar el contenido del fichero.
- Un programa que pregunte un nombre de fichero y muestre en pantalla el contenido de ese fichero, haciendo una pausa después de cada 25 líneas, para que dé tiempo a leerlo. Cuando el usuario pulse intro, se mostrarán las siguientes 25 líneas, y así hasta que termine el fichero.

7.4. Añadir a un fichero existente

La idea es sencilla: en vez de abrirlo con CreateText ("crear texto"), usaremos AppendText ("añadir texto"). Por ejemplo, podríamos crear un fichero, guardar información, cerrarlo, y luego volverlo a abrir para añadir datos, de la siguiente forma:

```
/*-----*/
/* Ejemplo en C# nº 73      */
/* ejemplo73.cs            */
/*
/* Añadir en un fichero    */
/* de texto                */
/*
/* Introducción a C#,       */
/* Nacho Cabanes          */
/*-----*/
```

```
using System;
using System.IO; // Para StreamWriter

public class Ejemplo73
{
    public static void Main()
    {
        StreamWriter fichero;

        fichero = File.CreateText("prueba2.txt");
        fichero.WriteLine("Primera línea");
        fichero.Close();

        fichero = File.AppendText("prueba2.txt");
        fichero.WriteLine("Segunda línea");
        fichero.Close();
    }
}
```

Ejercicios propuestos:

- Un programa que pida al usuario que teclee frases, y las almacene en el fichero "registro.txt", que puede existir anteriormente (y que no deberá borrarse, sino añadir al

final de su contenido). Cada sesión acabará cuando el usuario pulse Intro sin teclear nada.

-

7.5. Ficheros en otras carpetas

Si un fichero al que queremos acceder se encuentra en otra carpeta, basta con que indiquemos la ruta hasta ella, recordando que, como la barra invertida que se usa en sistemas Windows para separar los nombres de los directorios, coincide con el carácter de control que se usa en las cadenas de C y los lenguajes que derivan de él, deberemos escribir dichas barras invertidas repetidas, así:

```
string nombreFichero = "d:\\ejemplos\\ejemplo1.txt"; // Ruta absoluta
string nombreFichero2 = "..\\datos\\configuracion.txt"; // Ruta relativa
```

Como esta sintaxis puede llegar a resultar incómoda, en C# existe una alternativa: podemos indicar una arroba (@) justo antes de abrir las comillas, y entonces no será necesario delimitar los caracteres de control:

```
string nombreFichero = @"d:\\ejemplos\\ejemplo1.txt";
```

Ejercicios propuestos:

- Crear un programa que pida al usuario pares de números enteros y escriba su suma (con el formato "20 + 3 = 23") en pantalla y en un fichero llamado "sumas.txt", que se encontrará en un subdirectorio llamado "resultados". Cada vez que se ejecute el programa, deberá añadir los nuevos resultados a continuación de los resultados de las ejecuciones anteriores.

7.6. Saber si un fichero existe

Hasta ahora, estamos intentando abrir ficheros para lectura, pero sin comprobar realmente si el fichero existe o no, lo que puede suponer que nuestro programa falle en caso de que el fichero no se encuentre donde nosotros esperamos.

Una primera solución es usar "File.Exists(nombre)", para comprobar si está, antes de intentar abrirlo:

```
/*-----*/
/* Ejemplo en C# nº 74:      */
/* ejemplo74.cs              */
/*                           */
/* Lectura de un fichero de  */
/* texto                      */
/*                           */
/* Introducción a C#,          */
/* Nacho Cabanes             */
/*-----*/
```

```
using System;
using System.IO;

public class Ejemplo74
```

```

{
    public static void Main()
    {
        StreamReader fichero;
        string nombre;

        while (true) {
            Console.WriteLine("Dime el nombre del fichero (\\"fin\\" para terminar): ");
            nombre = Console.ReadLine();
            if (nombre == "fin")
                break;
            if (File.Exists(nombre) )
            {
                fichero = File.OpenText( nombre );
                Console.WriteLine("Su primera linea es: {0}",
                    fichero.ReadLine() );
                fichero.Close();
            }
            else
                Console.WriteLine("No existe!");
        }
    }
}

```

Ejercicios propuestos:

- Mejorar el segundo ejercicio del apartado 7.3 (el que muestra un fichero, haciendo una pausa cada 25 líneas) para que compruebe antes si el fichero existe, y muestre un mensaje de aviso en caso de que no sea así.

7.7. Más comprobaciones de errores: excepciones

El uso de "File.Exists" nos permite saber si el fichero existe, pero ese no es el único problema que podemos tener al acceder a un fichero. Puede ocurrir que no tengamos permiso para acceder al fichero, a pesar de que exista, o que intentemos escribir en un dispositivo que sea sólo de lectura (como un CD-ROM, por ejemplo).

Una forma más eficaz de comprobar si ha existido algún tipo de error es utilizar el manejo de "excepciones" que permiten los lenguajes modernos, como C#.

La idea es la siguiente: "intentaremos" dar una serie de pasos, y al final de todos ellos indicaremos qué pasos hay que dar en caso de que alguno no se consiga completar. Esto permite que el programa sea más legible que la alternativa "convencional", que consistía en: intentar un paso y comprobar errores; si todo era correcto, intentar otro paso y volver a comprobar errores; si todo seguía siendo correcto, intentar otro nuevo paso, y así sucesivamente.

La forma de conseguirlo es dividir en dos bloques las partes de programa que puedan dar lugar a error:

- En un primer bloque, indicaremos los pasos que queremos "intentar" (try).
- A continuación, detallaremos las posibles excepciones que queremos "interceptar" (catch), y lo que se debe hacer en ese caso.

Un primer ejemplo, que mostrara todo el contenido de un fichero de texto, y que en caso de error, se limitara a mostrar un mensaje de error y a abandonar el programa, podría ser:

```
/*
/* Ejemplo en C# nº 75: */
/* ejemplo75.cs */
/*
/* Excepciones y ficheros */
/* (1) */
/*
/* Introduccion a C#,
/* Nacho Cabanes */
/*
using System;
using System.IO;

public class Ejemplo75
{
    public static void Main()
    {
        StreamReader fichero;
        string nombre;
        string linea;

        Console.WriteLine("Introduzca el nombre del fichero");
        nombre = Console.ReadLine();

        try
        {
            fichero = File.OpenText(nombre);
            do
            {
                linea = fichero.ReadLine();
                if (linea != null)
                    Console.WriteLine( linea );
            }
            while (linea != null);

            fichero.Close();
        }
        catch (Exception exp)
        {
            Console.WriteLine("Ha habido un error: {0}", exp.Message);
            return;
        }
    }
}
```

El resultado, si ese fichero no existe, sería

```
Introduzca el nombre del fichero
prueba
Ha habido un error: No se pudo encontrar el archivo 'C:\Fuentes\nacho\prueba'.
```

Pero en general, lo razonable no es interceptar "todas las excepciones a la vez", sino crear un análisis para cada caso, que permita recuperarse del error y seguir adelante, para lo que se suelen crear varios bloques "catch". Por ejemplo, en el caso de que queramos crear un fichero, podemos tener excepciones como éstas:

- El fichero existe y es de sólo lectura (se lanzará una excepción del tipo "IOException").
- La ruta del fichero es demasiado larga (excepción de tipo "PathTooLongException").
- El disco puede estar lleno (IOException).

Así, dentro de cada bloque "catch" podríamos indicar una excepción más concreta, de forma que el mensaje de aviso sea más concreto, o que podamos dar pasos más adecuados para solucionar el problema:

```
/*
 *----- Ejemplo en C# nº 76: -----
 *----- ejemplo76.cs -----
 *----- -----
 *----- Excepciones y ficheros -----
 *----- (2) -----
 *----- -----
 *----- Introducción a C#, -----
 *----- Nacho Cabanes -----
 *----- -----*/
```

```
using System;
using System.IO;

public class Ejemplo76
{
    public static void Main()
    {
        StreamWriter fichero;
        string nombre;
        string linea;

        Console.WriteLine("Introduzca el nombre del fichero: ");
        nombre = Console.ReadLine();
        Console.WriteLine("Introduzca la frase a guardar: ");
        linea = Console.ReadLine();

        try
        {
            fichero = File.CreateText(nombre);
            fichero.WriteLine( linea );
            fichero.Close();
        }
        catch (PathTooLongException e)
        {
            Console.WriteLine("Nombre demasiado largo!");
        }
        catch (IOException e)
        {
```

```

        Console.WriteLine("No se ha podido escribir!");
        Console.WriteLine("El error exacto es: {0}", e.Message);
    }
}

```

Como la consola se comporta como un fichero de texto (realmente, como un fichero de entrada y otro de salida), se puede usar "try...catch" para comprobar ciertos errores relacionados con la entrada de datos, como cuando no se puede convertir un dato a un cierto tipo (por ejemplo, si queremos convertir a Int32, pero el usuario ha tecleado sólo texto).

Ejercicios propuestos:

- Un programa que pida al usuario el nombre de un fichero de origen y el de un fichero de destino, y que vuelque al segundo fichero el contenido del primero, convertido a mayúsculas. Se debe controlar los posibles errores, como que el fichero de origen no exista, o que el fichero de destino no se pueda crear.
- Un programa que pida al usuario un número, una operación (+, -, *, /) y un segundo número, y muestre el resultado de la correspondiente operación. Si se teclea un dato no numérico, el programa deberá mostrar un aviso y volver a pedirlo, en vez de interrumpir la ejecución.
- Un programa que pida al usuario repetidamente pares de números y la operación a realizar con ellos (+, -, *, /) y guarde en un fichero "calculadora.txt" el resultado de dichos cálculos (con la forma "15 * 6 = 90"). Debe controlar los posibles errores, como que los datos no sean numéricos, la división entre cero, o que el fichero no se haya podido crear.

7.8. Conceptos básicos sobre ficheros

Llega el momento de ver algunos conceptos que hemos pasado por encima, y que es necesario conocer:

- Fichero físico. Es un fichero que existe realmente en el disco, como "agenda.txt".
- Fichero lógico. Es un identificador que aparece dentro de nuestro programa, y que permite acceder a un fichero físico. Por ejemplo, si declaramos "StreamReader fichero1", esa variable "fichero1" representa un fichero lógico.
- Equivalencia entre fichero lógico y fichero físico. No necesariamente tiene por qué existir una correspondencia "1 a 1": puede que accedamos a un fichero físico mediante dos o más ficheros lógicos (por ejemplo, un StreamReader para leer de él y un StreamWriter para escribir en él), y también podemos usar un único fichero lógico para acceder a distintos ficheros físicos (por ejemplo, los mapas de los niveles de un juego, que estén guardados en distintos ficheros físicos, pero los abramos y los leamos utilizando siempre una misma variable).
- Registro: Un bloque de datos que forma un "todo", como el conjunto de los datos de una persona: nombre, dirección, e-mail, teléfono, etc. Cada uno de esos "apartados" de un registro se conoce como "campo".

- Acceso secuencial: Cuando debemos "recorrer" todo el contenido de un fichero si queremos llegar hasta cierto punto (como ocurre con las cintas de video o de casete). Es lo que estamos haciendo hasta ahora en los ficheros de texto.
- Acceso aleatorio: Cuando podemos saltar hasta cualquier posición del fichero directamente, sin necesidad de recorrer todo lo anterior. Es algo que comenzaremos a hacer pronto.

7.9. Leer datos básicos de un fichero binario

Los ficheros de texto son habituales, pero es aún más frecuente encontrarnos con ficheros en los que la información está estructurada como una secuencia de bytes, más o menos ordenada. Esto ocurre en las imágenes, los ficheros de sonido, de video, etc.

Vamos a ver cómo leer de un fichero "general", y lo aplicaremos a descifrar la información almacenada en ciertos formatos habituales, como una imagen BMP o un sonido MP3.

Como primer acercamiento, vamos a ver cómo abrir un fichero (no necesariamente de texto) y leer el primer byte que contiene. Usaremos una clase específicamente diseñada para leer datos de los tipos básicos existentes en C# (byte, int, float, etc.), la clase "BinaryReader":

```
/*-----*/
/* Ejemplo en C# nº 77:      */
/* ejemplo77.cs               */
/*-----*/
/* Ficheros binarios (1)      */
/*-----*/
/* Introduccion a C#,          */
/* Nacho Cabanes              */
/*-----*/
```

```
using System;
using System.IO;

public class Ejemplo77
{
    public static void Main()
    {
        BinaryReader fichero;
        string nombre;
        byte unDato;

        Console.WriteLine("Introduzca el nombre del fichero");
        nombre = Console.ReadLine();

        try
        {
            fichero = new BinaryReader(
                File.Open(nombre, FileMode.Open));
            unDato = fichero.ReadByte();
            Console.WriteLine("El byte leido es un {0}",
                unDato);
            fichero.Close();
        }
    }
}
```

```

        }
        catch (Exception exp)
        {
            Console.WriteLine(exp.Message);
            return;
        }
    }
}

```

La forma de abrir un BinaryReader es algo más incómoda que con los ficheros de texto, porque nos obliga a llamar a un constructor y a indicarle ciertas opciones sobre el modo de fichero (la habitual será simplemente "abrirlo", con " FileMode.Open"), pero a cambio podemos leer cualquier tipo de dato, no sólo texto: ReadByte lee un dato "byte", ReadInt32 lee un "int", ReadSingle lee un "float", ReadString lee un "string", etc.

Ejercicios propuestos:

- Abrir un fichero con extensión EXE y comprobar si realmente se trata de un ejecutable, mirando si los dos primeros bytes del fichero corresponden a una letra "M" y una letra "Z", respectivamente.

7.10. Leer bloques de datos de un fichero binario

Leer byte a byte (o float a float) puede ser cómodo, pero también es lento. Por eso, en la práctica es más habitual leer grandes bloques de datos. Típicamente, esos datos se guardarán como un array de bytes.

Para eso, usaremos la clase "FileStream", más genérica. Esta clase tiene método "Read", que nos permite leer una cierta cantidad de datos desde el fichero. Le indicaremos en qué array guardar esos datos, a partir de qué posición del array debe introducir los datos, y qué cantidad de datos se deben leer. Nos devuelve un valor, que es la cantidad de datos que se han podido leer realmente (porque puede ser menos de lo que le hemos pedido, si hay un error o estamos al final del fichero).

Para abrir el fichero usaremos "OpenRead", como en este ejemplo:

```

/*
 *----- Ejemplo en C# nº 78:
 *----- ejemplo78.cs
 */
/*
 *----- Ficheros binarios (2)
 */
/*
 *----- Introduccion a C#,
 *----- Nacho Cabanes
 */
/*
 *-----*/
using System;
using System.IO;

public class Ejemplo78

```

```

{
    public static void Main()
    {
        FileStream fichero;
        string nombre;
        byte[] datos;
        int cantidadLeida;

        Console.WriteLine("Introduzca el nombre del fichero");
        nombre = Console.ReadLine();

        try
        {
            fichero = File.OpenRead(nombre);
            datos = new byte[10];
            int posicion = 0;
            int cantidadALeer = 10;
            cantidadLeida = fichero.Read(datos, posicion, cantidadALeer);

            if (cantidadLeida < 10)
                Console.WriteLine("No se han podido leer todos los datos!");
            else {
                Console.WriteLine("El primer byte leido es {0}", datos[0]);
                Console.WriteLine("El tercero es {0}", datos[2]);
            }
            fichero.Close();
        }
        catch (Exception exp)
        {
            Console.WriteLine(exp.Message);
            return;
        }
    }
}

```

Ejercicios propuestos:

- Abrir un fichero con extensión EXE y comprobar si realmente se trata de un ejecutable, mirando si los dos primeros bytes del fichero corresponden a una letra "M" y una letra "Z", respectivamente. Se deben leer ambos bytes a la vez, usando un array.

7.11. La posición en el fichero

La clase FileStream tiene también un método ReadByte, que permite leer un único byte. En ese caso (y también a veces en el caso de leer todo un bloque), habitualmente nos interesaría situarnos antes en la posición exacta en la que se encuentra el dato que nos interesa leer, y esto podemos conseguirlo mediante acceso aleatorio, sin necesidad de leer todos los bytes anteriores.

Para ello, tenemos el método "Seek". A este método se le indican dos parámetros: la posición a la que queremos saltar, y el punto desde el que queremos que se cuente esa posición (desde el comienzo del fichero –SeekOrigin.Begin–, desde la posición actual –SeekOrigin.Current– o desde

el final del fichero –`SeekOrigin.End`–). La posición es un `Int64`, porque puede ser un número muy grande e incluso un número negativo (si miramos desde el final del fichero, porque desde él habrá que retroceder).

De igual modo, podemos saber en qué posición del fichero nos encontramos, consultando la propiedad "Position", así como la longitud del fichero, mirando su propiedad "Length", como en este ejemplo:

```
/*-----*/
/* Ejemplo en C# nº 79:      */
/* ejemplo79.cs               */
/*                           */
/* Ficheros binarios (3)    */
/*                           */
/* Introduccion a C#,        */
/* Nacho Cabanes             */
/*-----*/

using System;
using System.IO;

public class Ejemplo79
{
    public static void Main()
    {
        FileStream fichero;
        string nombre;
        byte[] datos;
        int cantidadLeida;

        Console.WriteLine("Introduzca el nombre del fichero");
        nombre = Console.ReadLine();

        try
        {
            fichero = File.OpenRead(nombre);
            datos = new byte[10];
            int posicion = 0;
            int cantidadALeer = 10;
            cantidadLeida = fichero.Read(datos, posicion, cantidadALeer);

            if (cantidadLeida < 10)
                Console.WriteLine("No se han podido leer todos los datos!");
            else {
                Console.WriteLine("El primer byte leido es {0}", datos[0]);
                Console.WriteLine("El tercero es {0}", datos[2]);
            }

            if (fichero.Length > 30) {
                fichero.Seek(19, SeekOrigin.Begin);
                int nuevoDato = fichero.ReadByte();
                Console.WriteLine("El byte 20 es un {0}", nuevoDato);

                Console.WriteLine("La posición actual es {0}",
                    fichero.Position);
                Console.WriteLine("Y el tamaño del fichero es {0}",
                    fichero.Length);
            }
        }
    }
}
```

```
        fichero.Length);  
    }  
  
    fichero.Close();  
}  
  
catch (Exception exp)  
{  
    Console.WriteLine(exp.Message);  
    return;  
}  
}  
}
```

(Nota: existe una propiedad "CanSeek" que nos permite saber si el fichero que hemos abierto permite realmente que nos movamos a unas posiciones u otras).

7.12. Escribir en un fichero binario

Para escribir en un FileStream, usaremos la misma estructura que para leer de él:

- Un método Write, para escribir un bloque de información (desde cierta posición de un array, y con cierto tamaño).
 - Un método WriteByte, para escribir sólo un byte.

Además, a la hora de abrir el fichero, tenemos dos alternativas:

- Abrir un fichero existente con "OpenWrite".
 - Crear un nuevo fichero con "Create".

Vamos a ver un ejemplo que junte todo ello: crearemos un fichero, guardaremos datos, lo leeremos para comprobar que todo es correcto, añadiremos al final, y volveremos a leer:

```
/*-----*/
/* Ejemplo en C# nº 80: */
/* ejemplo80.cs */
/*
/* Ficheros binarios (4):
/* Escritura
/*
/* Introduccion a C#,
/* Nacho Cabanes
/*
/*-----*/
```

```
using System;
using System.IO;

public class Ejemplo80
{
    const int TAMANO_BUFFER = 10;

    public static void Main()
    {
        FileStream fichero;
        string nombre;
```

```

byte[] datos;

nombre = "datos.dat";
datos = new byte[TAMANYO_BUFFER];

// Damos valores iniciales al array
for (byte i=0; i<TAMANYO_BUFFER; i++)
    datos[i] = (byte) (i + 10);

try
{
    int posicion = 0;

    // Primero creamos el fichero, con algun dato
    fichero = File.Create( nombre );
    fichero.Write(datos, posicion, TAMANYO_BUFFER);
    fichero.Close();

    // Ahora leemos dos datos
    fichero = File.OpenRead(nombre);
    Console.WriteLine("El tamaño es {0}", fichero.Length);
    fichero.Seek(2, SeekOrigin.Begin);
    int nuevoDato = fichero.ReadByte();
    Console.WriteLine("El tercer byte es un {0}", nuevoDato);
    fichero.Seek(-2, SeekOrigin.End);
    nuevoDato = fichero.ReadByte();
    Console.WriteLine("El penultimo byte es un {0}", nuevoDato);
    fichero.Close();

    // Ahora añadimos 10 datos más, al final
    fichero = File.OpenWrite(nombre);
    fichero.Seek(0, SeekOrigin.End);
    fichero.Write(datos, 0, TAMANYO_BUFFER);
    // y modificamos el tercer byte
    fichero.Seek(2, SeekOrigin.Begin);
    fichero.WriteByte( 99 );
    fichero.Close();

    // Volvemos a leer algun dato
    fichero = File.OpenRead(nombre);
    Console.WriteLine("El tamaño es {0}", fichero.Length);
    fichero.Seek(2, SeekOrigin.Begin);
    nuevoDato = fichero.ReadByte();
    Console.WriteLine("El tercer byte es un {0}", nuevoDato);
    fichero.Close();
}

catch (Exception exp)
{
    Console.WriteLine(exp.Message);
    return;
}
}
}

```

(Nota: existe una propiedad "CanWrite" que nos permite saber si se puede escribir en el fichero).

Si queremos que escribir datos básicos de C# (float, int, etc.) en vez de un array de bytes, podemos usar un "BinaryWriter", que se maneja de forma similar a un "BinaryReader", con la diferencia de que no tenemos métodos WriteByte, WriteString y similares, sino un único método "Write", que se encarga de escribir el dato que le indiquemos, sea del tipo que sea:

```
/*-----*/
/* Ejemplo en C# nº 81:      */
/* ejemplo81.cs               */
/*                           */
/* Ficheros binarios (5)     */
/*                           */
/* Introduccion a C#,        */
/* Nacho Cabanes             */
/*-----*/
using System;
using System.IO;

public class Ejemplo81
{
    public static void Main()
    {
        BinaryWriter ficheroSalida;
        BinaryReader ficheroEntrada;
        string nombre;
        // Los datos que vamos a guardar/leer
        byte unDatoByte;
        int unDatoInt;
        float unDatoFloat;
        double unDatoDouble;
        string unDatoString;

        Console.Write("Introduzca el nombre del fichero a crear: ");
        nombre = Console.ReadLine();

        Console.WriteLine("Creando fichero...");
        // Primero vamos a grabar datos
        try
        {
            ficheroSalida = new BinaryWriter(
                File.Open(nombre, FileMode.Create));
            unDatoByte = 1;
            unDatoInt = 2;
            unDatoFloat = 3.0f;
            unDatoDouble = 4.0;
            unDatoString = "Hola";
            ficheroSalida.Write(unDatoByte);
            ficheroSalida.Write(unDatoInt);
            ficheroSalida.Write(unDatoFloat);
            ficheroSalida.Write(unDatoDouble);
            ficheroSalida.Write(unDatoString);
            ficheroSalida.Close();
        }
    }
}
```

```

        }
        catch (Exception exp)
        {
            Console.WriteLine(exp.Message);
            return;
        }

        // Ahora vamos a volver a leerlos
        Console.WriteLine("Leyendo fichero...");
        try
        {
            ficheroEntrada = new BinaryReader(
                File.Open(nombre, FileMode.Open));
            unDatoByte = ficheroEntrada.ReadByte();
            Console.WriteLine("El byte leido es un {0}",
                unDatoByte);
            unDatoInt = ficheroEntrada.ReadInt32();
            Console.WriteLine("El int leido es un {0}",
                unDatoInt);
            unDatoFloat = ficheroEntrada.ReadSingle();
            Console.WriteLine("El float leido es un {0}",
                unDatoFloat);
            unDatoDouble = ficheroEntrada.ReadDouble();
            Console.WriteLine("El double leido es un {0}",
                unDatoDouble);
            unDatoString = ficheroEntrada.ReadString();
            Console.WriteLine("El string leido es \"{0}\"",
                unDatoString);
            Console.WriteLine("Volvamos a leer el int...");
            ficheroEntrada.BaseStream.Seek(1, SeekOrigin.Begin);
            unDatoInt = ficheroEntrada.ReadInt32();
            Console.WriteLine("El int leido es un {0}",
                unDatoInt);
            ficheroEntrada.Close();
        }
        catch (Exception exp)
        {
            Console.WriteLine(exp.Message);
            return;
        }
    }
}

```

Como se puede ver en este ejemplo, también podemos usar "Seek" para movernos a un punto u otro de un fichero si usamos un "BinaryReader", pero está un poco más escondido: no se lo pedimos directamente a nuestro fichero, sino al "Stream" (flujo de datos) que hay por debajo: `ficheroEntrada.BaseStream.Seek(1, SeekOrigin.Begin);`

El resultado de este programa es:

```

Introduzca el nombre del fichero a crear: 1234
Creando fichero...
Leyendo fichero...
El byte leido es un 1
El int leido es un 2

```

```

El float leido es un 3
El double leido es un 4
El string leido es "Hola"
Volvamos a leer el int...
El int leido es un 2

```

En este caso hemos usado " FileMode.Create" para indicar que queremos crear el fichero, en vez de abrir un fichero ya existente. Los modos de fichero que podemos emplear en un BinaryReader o en un BinaryWriter son los siguientes:

- CreateNew: Crear un archivo nuevo. Si existe, se produce una excepción IOException.
- Create: Crear un archivo nuevo. Si ya existe, se sobrescribirá.
- Open: Abrir un archivo existente. Si el archivo no existe, se produce una excepción System.IO.FileNotFoundException.
- OpenOrCreate: Se debe abrir un archivo si ya existe; en caso contrario, debe crearse uno nuevo.
- Truncate: Abrir un archivo existente y truncarlo para que su tamaño sea de cero bytes.
- Append: Abre el archivo si existe y realiza una búsqueda hasta el final del mismo, o crea un archivo nuevo si no existe.

7.13. Ejemplo: leer información de un fichero BMP

Ahora vamos a ver un ejemplo un poco más sofisticado y un poco más real: vamos a abrir un fichero que sea una imagen en formato BMP y a mostrar en pantalla si está comprimido o no.

Para eso necesitamos antes saber cómo se guarda la información en un fichero BMP, pero esto es algo fácil de localizar en Internet:

FICHEROS .BMP

Un fichero BMP está compuesto por las siguientes partes: una cabecera de fichero, una cabecera del bitmap, una tabla de colores y los bytes que definirán la imagen.

En concreto, los datos que forman la cabecera de fichero y la cabecera de bitmap son los siguientes:

TIPO DE INFORMACIÓN	POSICIÓN EN EL ARCHIVO
Tipo de fichero (letras BM)	0-1
Tamaño del archivo	2-5
Reservado	6-7
Reservado	8-9
Inicio de los datos de la imagen	10-13
Tamaño de la cabecera de bitmap	14-17
Anchura (píxeles)	18-21
Altura (píxeles)	22-25
Número de planos	26-27
Tamaño de cada punto	28-29

Compresión (0=no comprimido)	30-33
Tamaño de la imagen	34-37
Resolución horizontal	38-41
Resolución vertical	42-45
Tamaño de la tabla de color	46-49
Contador de colores importantes	50-53

Con esta información nos basta para nuestro propósito: la compresión se indica en la posición 30 del fichero, es un entero de 4 bytes (lo mismo que un "int" en los sistemas operativos de 32 bits), y si es un 0 querrá decir que la imagen no está comprimida.

Como el bit menos significativo se almacena en primer lugar, nos podría bastar con leer sólo el byte de la posición 30, para ver si vale 0, y despreciar los 3 bytes siguientes. Entonces, lo podríamos comprobar así:

```
/*
 * Ejemplo en C# nº 82:
 */
/* ejemplo82.cs */
/*
 */
/* Ficheros binarios (6): */
/* Ver si un BMP está */
/* comprimido */
/*
 */
/* Introducción a C#, */
/* Nacho Cabanes */
/*
 */

using System;
using System.IO;

public class Ejemplo82
{
    public static void Main()
    {
        FileStream fichero;
        string nombre;
        int compresion;

        Console.WriteLine("Comprobador de imágenes BMP\n");
        Console.WriteLine("Dime el nombre del fichero: ");
        nombre = Console.ReadLine();

        if (! File.Exists( nombre ) )
        {
            Console.WriteLine("No encontrado!");
        }
        else
        {
            fichero = File.OpenRead(nombre);
            fichero.Seek(30, SeekOrigin.Begin);
            compresion = fichero.ReadByte();
            fichero.Close();

            if (compresion == 0)

```

```
        Console.WriteLine("Sin compresión");  
    else  
        Console.WriteLine("BMP Comprimido ");  
    }  
}  
}
```

Ya que estamos, podemos mejorarla un poco para que además nos muestre el ancho y el alto de la imagen, y que compruebe antes si realmente se trata de un fichero BMP:

```
/*
/* Ejemplo en C# nº 83:      */
/* ejemplo83.cs                */
/*
/* Ficheros binarios (7):    */
/* Información de un          */
/* fichero BMP                  */
/*
/* Introducción a C#,          */
/* Nacho Cabanes               */
/*-----*/
```

using System;
using System.IO;

```
public class Ejemplo83
{
    public static void Main()
    {
        FileStream fichero;
        string nombre;
        int compresion, ancho, alto;
        char marca1, marca2;
        byte[] datosTemp = new byte[4];

        Console.WriteLine("Comprobador de imágenes BMP\n");
        Console.WriteLine("Dime el nombre del fichero: ");
        nombre = Console.ReadLine();

        if (! File.Exists( nombre ) )
        {
            Console.WriteLine("No encontrado!");
        }
        else
        {
            fichero = File.OpenRead(nombre);
            // Leo los dos primeros bytes
            marca1 = Convert.ToChar( fichero.ReadByte() );
            marca2 = Convert.ToChar( fichero.ReadByte() );

            if ((marca1 == 'B') && (marca2 == 'M')) { // Si son BM
                Console.WriteLine("Marca del fichero: {0}{1}",
                    marca1, marca2);

                fichero.Seek(18, SeekOrigin.Begin); // Posición 18: ancho
            }
        }
    }
}
```

```

        fichero.Read(datosTemp, 0, 4);
        ancho = datosTemp[0] +           // Convierto 4 bytes a Int32
            datosTemp[1] * 256 + datosTemp[2] * 256 * 256
            + datosTemp[3] * 256 * 256 * 256;
        Console.WriteLine("Ancho: {0}", ancho);

        fichero.Read(datosTemp, 0, 4);
        alto = datosTemp[0] +           // Convierto 4 bytes a Int32
            datosTemp[1] * 256 + datosTemp[2] * 256 * 256
            + datosTemp[3] * 256 * 256 * 256;
        Console.WriteLine("Alto: {0}", alto);

        fichero.Seek(4, SeekOrigin.Current); // 4 bytes después: compresión
        compresion = fichero.ReadByte();
        fichero.Close();

        switch (compresion) {
            case 0: Console.WriteLine("Sin compresión"); break;
            case 1: Console.WriteLine("Compresión RLE 8 bits"); break;
            case 2: Console.WriteLine("Compresión RLE 4 bits"); break;
        }
    } else
        Console.WriteLine("No parece un fichero BMP\n"); // Si la marca no es BM
}
}
}

```

También podemos hacer lo mismo empleando un "BinaryReader", en lugar de un "FileStream". En ese caso, se simplifica la lectura de datos de 32 bits, a cambio de complicarse ligeramente la apertura y los "Seek", como se ve en este ejemplo:

```

/*
/* Ejemplo en C# nº 84:      */
/* ejemplo84.cs               */
/*                                */
/* Ficheros binarios (8):     */
/* Información de un BMP       */
/* con BinaryReader           */
/*                                */
/* Introducción a C#,          */
/* Nacho Cabanes              */
/*-----*/
using System;
using System.IO;

public class Ejemplo84
{

    public static void Main()
    {
        BinaryReader fichero;
        string nombre;
        int compresion, ancho, alto;
        char marca1, marca2;

```

```

Console.WriteLine("Comprobador de imágenes BMP\n");
Console.WriteLine("Dime el nombre del fichero: ");
nombre = Console.ReadLine();

if (! File.Exists( nombre) )
{
    Console.WriteLine("No encontrado!");
}
else
{
    fichero = new BinaryReader(
        File.Open(nombre, FileMode.Open));
    // Leo los dos primeros bytes
    marca1 = Convert.ToChar( fichero.ReadByte() );
    marca2 = Convert.ToChar( fichero.ReadByte() );

    if ((marca1 =='B') && (marca2 =='M')) { // Si son BM
        Console.WriteLine("Marca del fichero: {0}{1}",
            marca1, marca2);

        fichero.BaseStream.Seek(18, SeekOrigin.Begin); // Posición 18: ancho
        ancho = fichero.ReadInt32();
        Console.WriteLine("Ancho: {0}", ancho);

        alto = fichero.ReadInt32();
        Console.WriteLine("Alto: {0}", alto);

        fichero.BaseStream.Seek(4, SeekOrigin.Current); // 4 bytes después: compresión
        compresion = fichero.ReadInt32();
        fichero.Close();

        switch (compresion) {
            case 0: Console.WriteLine("Sin compresión"); break;
            case 1: Console.WriteLine("Compresión RLE 8 bits"); break;
            case 2: Console.WriteLine("Compresión RLE 4 bits"); break;
        }
    } else
        Console.WriteLine("No parece un fichero BMP\n"); // Si la marca no es BM
}
}

```

Ejercicios propuestos:

- Localiza en Internet información sobre el formato de imágenes PCX. Crea un programa que diga el ancho, alto y número de colores de una imagen PCX.
- Localiza en Internet información sobre el formato de imágenes GIF. Crea un programa que diga el subformato, ancho, alto y número de colores de una imagen GIF.

7.14. Leer y escribir en un mismo fichero binario

También es posible que nos interese leer y escribir en un mismo fichero (por ejemplo, para poder modificar algún dato erróneo, o para poder crear un editor hexadecimal). Podemos conseguirlo abriendo (en modo de lectura o de escritura) o cerrando el fichero cada vez, pero

también tenemos la alternativa de usar un "FileStream", que también tiene un método llamado simplemente "Open", al que se le puede indicar el modo de apertura (FileMode, como se vieron en el apartado 7.12) y el modo de acceso (FileAccess.Read si queremos leer, FileAccess.Write si queremos escribir, o FileAccess.ReadWrite si queremos leer y escribir).

Una vez que hayamos indicado que queremos leer y escribir del fichero, podremos movernos dentro de él con "Seek", leer datos con "Read" o "ReadByte", y grabar datos con "Write" o "WriteByte":

```
/*
 * Ejemplo en C# nº 85:
 */
/* ejemplo85.cs */
/*
 */
/* Ficheros binarios (9):
 */
/* Lectura y Escritura */
/*
 */
/* Introducción a C#,
 */
/* Nacho Cabanes */
/*
 */

using System;
using System.IO;

public class Ejemplo85
{
    const int TAMANO_BUFFER = 10;

    public static void Main()
    {
        FileStream fichero;
        string nombre;
        byte[] datos;

        nombre = "datos.dat";
        datos = new byte[TAMANO_BUFFER];

        // Damos valores iniciales al array
        for (byte i=0; i<TAMANO_BUFFER; i++)
            datos[i] = (byte) (i + 10);

        try
        {
            int posicion = 0;

            // Primero creamos el fichero, con algún dato
            fichero = File.Create( nombre );
            fichero.Write(datos, posicion, TAMANO_BUFFER);
            fichero.Close();

            // Ahora leemos dos datos
            fichero = File.Open(nombre, FileMode.Open, FileAccess.ReadWrite);

            fichero.Seek(2, SeekOrigin.Begin);
            int nuevoDato = fichero.ReadByte();
            Console.WriteLine("El tercer byte es un {0}", nuevoDato);
        }
    }
}
```

```

fichero.Seek(2, SeekOrigin.Begin);
fichero.WriteByte(4);

fichero.Seek(2, SeekOrigin.Begin);
nuevoDato = fichero.ReadByte();
Console.WriteLine("Ahora el tercer byte es un {0}", nuevoDato);

fichero.Close();
}
catch (Exception exp)
{
    Console.WriteLine(exp.Message);
    return;
}
}

}

```

Ejercicios propuestos:

- Un programa que vuelque todo el contenido de un fichero de texto a otro, convirtiendo cada frase a mayúsculas. Los nombres de ambos ficheros se deben indicar como parámetros en la línea de comandos.
- Un programa que pida al usuario el nombre de un fichero de texto y una frase a buscar, y que muestre en pantalla todas las frases de ese fichero que contengan esa frase. Cada frase se debe preceder del número de línea (la primera línea del fichero será la 1, la segunda será la 2, y así sucesivamente).
- Un programa que pida al usuario el nombre de un fichero y una secuencia de 4 bytes, y diga si el fichero contiene esa secuencia de bytes.
- Un programa que duplique un fichero, copiando todo su contenido a un nuevo fichero. El nombre de ambos ficheros se debe leer de la línea de comandos.
- Un programa que muestre el nombre del autor de un fichero de música en formato MP3 (tendrás que localizar en Internet la información sobre dicho formato).

8. Punteros y gestión dinámica de memoria

8.1. ¿Por qué usar estructuras dinámicas?

Hasta ahora teníamos una serie de variables que declaramos al principio del programa o de cada función. Estas variables, que reciben el nombre de **ESTÁTICAS**, tienen un tamaño asignado desde el momento en que se crea el programa.

Este tipo de variables son sencillas de usar y rápidas... si sólo vamos a manejar estructuras de datos que no cambien, pero resultan poco eficientes si tenemos estructuras cuyo tamaño no sea siempre el mismo.

Es el caso de una agenda: tenemos una serie de fichas, e iremos añadiendo más. Si reservamos espacio para 10, no podremos llegar a añadir la número 11, estamos limitando el máximo. Una solución sería la de trabajar siempre en el disco: no tenemos límite en cuanto a número de fichas, pero es muchísimo más lento.

Lo ideal sería aprovechar mejor la memoria que tenemos en el ordenador, para guardar en ella todas las fichas o al menos todas aquellas que quepan en memoria.

Una solución "típica" (pero mala) es sobredimensionar: preparar una agenda contando con 1000 fichas, aunque supongamos que no vamos a pasar de 200. Esto tiene varios inconvenientes: se desperdicia memoria, obliga a conocer bien los datos con los que vamos a trabajar, sigue pudiendo verse sobrepasado, etc.

La solución suele ser crear estructuras **DINÁMICAS**, que puedan ir creciendo o disminuyendo según nos interese. En los lenguajes de programación "clásicos", como C y Pascal, este tipo de estructuras se tienen que crear de forma básicamente artesanal, mientras que en lenguajes modernos como C#, Java o las últimas versiones de C++, existen esqueletos ya creados que podemos utilizar con facilidad.

Algunos ejemplos de estructuras de este tipo son:

- Las **pilas**. Como una pila de libros: vamos apilando cosas en la cima, o cogiendo de la cima. Se supone que no se puede tomar elementos de otro sitio que no sea la cima, ni dejarlos en otro sitio distinto. De igual modo, se supone que la pila no tiene un tamaño máximo definido, sino que puede crecer arbitrariamente.
- Las **colas**. Como las del cine (en teoría): la gente llega por un sitio (la cola) y sale por el opuesto (la cabeza). Al igual que antes, supondremos que un elemento no puede entrar a la cola ni salir de ella en posiciones intermedias y que la cola puede crecer hasta un tamaño indefinido.
- Las **listas**, en las que se puede añadir elementos en cualquier posición, y borrarlos de cualquier posición.

Y la cosa se va complicando: en los **árboles** cada elemento puede tener varios sucesores (se parte de un elemento "raíz", y la estructura se va ramificando), etc.

Todas estas estructuras tienen en común que, si se programan correctamente, pueden ir creciendo o decreciendo según haga falta, al contrario que un array, que tiene su tamaño prefijado.

Veremos ejemplos de cómo crear estructuras dinámicas de estos tipos en C#, y después comentaremos los pasos para crear una estructura dinámica de forma "artesanal".

8.2. Una pila en C#

Para crear una pila tenemos preparada la clase Stack. Los métodos habituales que debería permitir una pila son introducir un nuevo elemento en la cima ("apilar", en inglés "push"), y quitar el elemento que hay en la cima ("desapilar", en inglés "pop"). Este tipo de estructuras se suele llamar también con las siglas "LIFO" (Last In First Out: lo último en entrar es lo primero en salir). Para utilizar la clase "Stack" y la mayoría de las que veremos en este tema, necesitamos incluir en nuestro programa una referencia a "System.Collections".

Así, un ejemplo básico que creara una pila, introdujera tres palabras y luego las volviera a mostrar sería:

```
/*-----*/
/* Ejemplo en C# */
/* pilal.cs */
/*
/* Ejemplo de clase "Stack" */
/*
/* Introduccion a C#,
/* Nacho Cabanes */
/*-----*/
using System;
using System.Collections;

public class ejemploPilal {
    public static void Main() {
        string palabra;

        Stack miPila = new Stack();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        for (byte i=0; i<3; i++) {
            palabra = (string) miPila.Pop();
            Console.WriteLine( palabra );
        }
    }
}
```

cuyo resultado sería:

```
yo
soy
Hola,
```

La implementación de una pila en C# es algo más avanzada: permite también métodos como:

- "Peek", que mira el valor que hay en la cima, pero sin extraerlo.
- "Clear", que borra todo el contenido de la pila.
- "Contains", que indica si un cierto elemento está en la pila.
- "GetType", para saber de qué tipo son los elementos almacenados en la pila.
- "ToString", que devuelve el elemento actual convertido a un string.
- "ToArray", que devuelve toda la pila convertida a un array.
- "GetEnumerator", que permite usar "enumeradores" para recorrer la pila, una funcionalidad que veremos con algún detalle más adelante.
- También tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

8.3. Una cola en C#

Podemos crear colas si nos apoyamos en la clase Queue. En una cola podremos introducir elementos por la cabeza ("Enqueue", encolar) y extraerlos por el extremo opuesto, el final de la cola ("Dequeue", desencolar). Este tipo de estructuras se nombran a veces también por las siglas FIFO (First In First Out, lo primero en entrar es lo primero en salir). Un ejemplo básico similar al anterior, que creara una cola, introdujera tres palabras y luego las volviera a mostrar sería:

```
/*-----*/
/* Ejemplo en C#
/* cola1.cs
/*
/* Ejemplo de clase "Queue"
/*
/* Introduccion a C#,
/* Nacho Cabanes
/*-----*/

using System;
using System.Collections;

public class ejemploCola1 {
    public static void Main() {
        string palabra;

        Queue miCola = new Queue();
        miCola.Enqueue("Hola,");
        miCola.Enqueue("soy");
        miCola.Enqueue("yo");

        for (byte i=0; i<3; i++) {
            palabra = (string) miCola.Dequeue();
            Console.WriteLine(palabra);
        }
    }
}
```

```

        Console.WriteLine( palabra );
    }

}

}

```

que mostraría:

```

Hola,
soy
yo

```

Al igual que ocurría con la pila, la implementación de una cola que incluye C# es más avanzada que eso, con métodos similares a los de antes:

- "Peek", que mira el valor que hay en la cabeza de la cola, pero sin extraerlo.
- "Clear", que borra todo el contenido de la cola.
- "Contains", que indica si un cierto elemento está en la cola.
- "GetType", para saber de qué tipo son los elementos almacenados en la cola.
- "ToString", que devuelve el elemento actual convertido a un string.
- "ToArray", que devuelve toda la pila convertida a un array.
- "GetEnumerator", que permite usar "enumeradores" para recorrer la cola, una funcionalidad que veremos con algún detalle más adelante.
- Al igual que en la pila, también tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

8.4. Las listas

Una lista es una estructura dinámica en la que se puede añadir elementos sin tantas restricciones. Es habitual que se puedan introducir nuevos datos en ambos extremos, así como entre dos elementos existentes, o bien incluso de forma ordenada, de modo que cada nuevo dato se introduzca automáticamente en la posición adecuada para que todos ellos queden en orden.

En el caso de C#, no tenemos ninguna clase "List" que represente una lista genérica, pero sí dos variantes especialmente útiles: una lista ordenada ("SortedList") y una lista a cuyos elementos se puede acceder como a los de un array ("ArrayList").

8.4.1. ArrayList

En un ArrayList, podemos añadir datos en la última posición con "Add", insertar en cualquier otra con "Insert", recuperar cualquier elemento usando corchetes, o bien ordenar toda la lista con "Sort". Vamos a ver un ejemplo de la mayoría de sus posibilidades:

```

/*
 * Ejemplo en C#
 */
/* arrayList1.cs */
/*
 * Ejemplo de ArrayList */

```

```

/*
 * Introducción a C#,
 * Nacho Cabanes
 */
using System;
using System.Collections;

public class ejemploArrayList1 {

    public static void Main() {

        ArrayList miLista = new ArrayList();
        // Añadimos en orden
        miLista.Add("Hola,");
        miLista.Add("soy");
        miLista.Add("yo");

        // Mostramos lo que contiene
        Console.WriteLine( "Contenido actual:" );
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Accedemos a una posición
        Console.WriteLine( "La segunda palabra es: {0}" ,
            miLista[1] );

        // Insertamos en la segunda posición
        miLista.Insert(1, "Como estas?" );

        // Mostramos de otra forma lo que contiene
        Console.WriteLine( "Contenido tras insertar:" );
        for (int i=0; i<miLista.Count; i++)
            Console.WriteLine( miLista[i] );

        // Buscamos un elemento
        Console.WriteLine( "La palabra \"yo\" está en la posición {0}" ,
            miLista.IndexOf("yo") );

        // Ordenamos
        miLista.Sort();

        // Mostramos lo que contiene
        Console.WriteLine( "Contenido tras ordenar:" );
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Buscamos con búsqueda binaria
        Console.WriteLine( "Ahora \"yo\" está en la posición {0}" ,
            miLista.BinarySearch("yo") );

        // Invertimos la lista
        miLista.Reverse();

        // Borramos el segundo dato y la palabra "yo"
        miLista.RemoveAt(1);
        miLista.Remove("yo");
    }
}

```

```

// Mostramos nuevamente lo que contiene
Console.WriteLine( "Contenido dar la vuelta y tras eliminar dos:" );
foreach ( string frase in miLista )
    Console.WriteLine( frase );

// Ordenamos y vemos dónde iría un nuevo dato
miLista.Sort();
Console.WriteLine( "La frase \"Hasta Luego\"..." );
int posicion = miLista.BinarySearch("Hasta Luego");
if ( posicion >= 0 )
    Console.WriteLine( "Está en la posición {0}", posicion );
else
    Console.WriteLine( "No está. El dato inmediatamente mayor es el {0}: {1}",
        ~posicion, miLista[~posicion] );

}

}

```

El resultado de este programa es:

```

Contenido actual:
Hola,
soy
yo
La segunda palabra es: soy
Contenido tras insertar:
Hola,
Como estas?
soy
yo
La palabra "yo" está en la posición 3
Contenido tras ordenar
Como estas?
Hola,
soy
yo
Ahora "yo" está en la posición 3
Contenido dar la vuelta y tras eliminar dos:
Hola,
Como estas?
La frase "Hasta Luego"...
No está. El dato inmediatamente mayor es el 1: Hola,

```

Casi todo debería resultar fácil de entender, salvo quizás el símbolo `~`. Esto se debe a que `BinarySearch` devuelve un número negativo cuando el texto que buscamos no aparece, pero ese número negativo tiene un significado: es el "valor complementario" de la posición del dato inmediatamente mayor (es decir, el dato cambiando los bits 0 por 1 y viceversa). En el ejemplo anterior, "posición" vale -2, lo que quiere decir que el dato no existe, y que el dato inmediatamente mayor está en la posición 1 (que es el "complemento a 2" del número -2, que es lo que indica la expresión `~posición`). En el apéndice 3 de este texto hablaremos de cómo

se representan internamente los números enteros, tanto positivos como negativos, y entonces se verá con detalle en qué consiste el "complemento a 2".

A efectos prácticos, lo que nos interesa es que si quisieramos insertar la frase "Hasta Luego", su posición correcta para que todo el ArrayList permaneciera ordenado sería la 1, que viene indicada por " \sim posición".

Veremos los operadores a nivel de bits, como \sim , en el tema 10, que estará dedicado a otras características avanzadas de C#.

8.4.2. SortedList

En un SortedList, los elementos están formados por una pareja: una clave y un valor (como en un diccionario: la palabra y su definición). Se puede añadir elementos con "Add", o acceder a los elementos mediante su índice numérico (con "GetKey") o mediante su clave (sabiendo en qué posición se encuentra una clave con "IndexOfKey"), como en este ejemplo:

```
/*
 *----- Ejemplo en C#
 *----- sortedList1.cs
 */
/*
 *----- Ejemplo de SortedList: Diccionario esp-ing
 */
/*
 *----- Introduccion a C#, Nacho Cabanes
 */
/*-----*/
```

```
using System;
using System.Collections;
public class ejemploSortedList {

    public static void Main() {

        // Creamos e insertamos datos
        SortedList miDiccio = new SortedList();
        miDiccio.Add("hola", "hello");
        miDiccio.Add("adiós", "good bye");
        miDiccio.Add("hasta luego", "see you later");

        // Mostramos los datos
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}" ,
            miDiccio.Count );
        Console.WriteLine( "Lista de palabras y su significado:" );
        for (int i=0; i<miDiccio.Count; i++) {
            Console.WriteLine( "{0} = {1}" ,
                miDiccio.GetKey(i), miDiccio.GetByIndex(i) );
        }
        Console.WriteLine( "Traducción de \"hola\": {0}" ,
            miDiccio.GetByIndex( miDiccio.IndexOfKey("hola") ) );
    }
}
```

Su resultado sería

```
Cantidad de palabras en el diccionario: 3
Lista de palabras y su significado:
adiós = good bye
hasta luego = see you later
hola = hello
Traducción de "hola": hello
```

Otras posibilidades de la clase SortedList son:

- "Contains", para ver si la lista contiene una cierta clave.
- "ContainsValue", para ver si la lista contiene un cierto valor.
- "Remove", para eliminar un elemento a partir de su clave.
- "RemoveAt", para eliminar un elemento a partir de su posición.
- "SetByIndex", para cambiar el valor que hay en una cierta posición.

8.5. Las "tablas hash"

En una "tabla hash", los elementos están formados por una pareja: una clave y un valor, como en un SortedList, pero la diferencia está en la forma en que se manejan internamente estos datos: la "tabla hash" usa una "función de dispersión" para colocar los elementos, de forma que no se pueden recorrer secuencialmente, pero a cambio el acceso a partir de la clave es muy rápido, más que si hacemos una búsqueda secuencial (como en un array) o binaria (como en un ArrayList ordenado).

Un ejemplo de diccionario, parecido al anterior (que es más rápido de consultar para un dato concreto, pero que no se puede recorrer en orden), podría ser:

```
/*
 *----- Ejemplo en C#
 *----- HashTable1.cs
 *----- Ejemplo de HashTable:
 *----- Diccionario de inform.
 *----- Introduccion a C#,
 *----- Nacho Cabanes
 *----- */

using System;
using System.Collections;
public class ejemploHashTable {

    public static void Main() {

        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
    }
}
```

```

    miDiccio.Add("kilobyte", "1024 bytes");

    // Mostramos algún dato
    Console.WriteLine( "Cantidad de palabras en el diccionario: {0}",
        miDiccio.Count );
    try {
        Console.WriteLine( "El significado de PC es: {0}",
            miDiccio["pc"]);
    } catch (Exception e) {
        Console.WriteLine( "No existe esa palabra!" );
    }
}

}

```

que escribiría en pantalla:

```

Cantidad de palabras en el diccionario: 3
El significado de PC es: personal computer

```

Si un elemento que se busca no existe, se lanzaría una excepción, por lo que deberíamos controlarlo con un bloque try..catch. Lo mismo ocurre si intentamos introducir un dato que ya existe. Una alternativa a usar try..catch es comprobar si el dato ya existe, con el método "Contains" (o su sinónimo "ContainsKey"), como en este ejemplo:

```

/*
 * Ejemplo en C#
 */
/*
 * HashTable2.cs
 */
/*
 * Ejemplo de HashTable 2:
 */
/*
 * Diccionario de inform.
 */
/*
 * Introduccion a C#,
 */
/*
 * Nacho Cabanes
 */
/*-----*/
using System;
using System.Collections;
public class ejemploHashTable2 {

    public static void Main() {

        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos algún dato
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}",
            miDiccio.Count );
        if (miDiccio.Contains("pc"))
            Console.WriteLine( "El significado de PC es: {0}",
                miDiccio["pc"]);
        else

```

```

        Console.WriteLine( "No existe la palabra PC");
    }
}

```

Otras posibilidades son: borrar un elemento ("Remove"), vaciar toda la tabla ("Clear"), o ver si contiene un cierto valor ("ContainsValue", mucho más lento que buscar entre las claves con "Contains").

Una tabla hash tiene una cierta capacidad inicial, que se amplía automáticamente cuando es necesario. Como la tabla hash es mucho más rápida cuando está bastante vacía que cuando está casi llena, podemos usar un constructor alternativo, en el que se le indica la capacidad inicial que queremos, si tenemos una idea aproximada de cuántos datos vamos a guardar:

```
Hashtable miDiccio = new Hashtable(500);
```

8.6. Los "enumeradores"

Un enumerador es una estructura auxiliar que permite recorrer las estructuras dinámicas de forma secuencial. Casi todas ellas contienen un método GetEnumerator, que permite obtener un enumerador para recorrer todos sus elementos. Por ejemplo, en una tabla hash podríamos hacer:

```

/*
 * Ejemplo en C#
 */
/* HashTable3.cs
 */
/*
 * Ejemplo de HashTable
 */
/* y enumerador
 */
/*
 * Introduccion a C#,
 */
/* Nacho Cabanes
 */
/*
 */

using System;
using System.Collections;
public class ejemploHashTable3 {

    public static void Main() {

        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IDictionaryEnumerator miEnumerador = miDiccio.GetEnumerator();
        while ( miEnumerador.MoveNext() )
            Console.WriteLine("{0} = {1}",
                miEnumerador.Key, miEnumerador.Value);
    }
}

```

```
 }
```

cuyo resultado es

Contenido:

```
pc = personal computer
byte = 8 bits
kilobyte = 1024 bytes
```

Como se puede ver, los enumeradores tendrán un método "MoveNext", que intenta moverse al siguiente elemento y devuelve "false" si no lo consigue. En el caso de las tablas hash, que tiene dos campos (clave y valor), el enumerador a usar será un "enumerador de diccionario" (IDictionaryEnumerator), que contiene los campos Key y Value.

Como se ve en el ejemplo, es habitual que no obtengamos la lista de elementos en el mismo orden en el que los introdujimos, debido a que se colocan siguiendo la función de dispersión.

Para las colecciones "normales", como las pilas y las colas, el tipo de Enumerador a usar será un IEnumerator, con un campo Current para saber el valor actual:

```
/*
 * Ejemplo en C#
 */
/* pila2.cs */
/*
 */
/* Ejemplo de clase "Stack" */
/* y enumerador */
/*
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes */
/*
 */

using System;
using System.Collections;

public class ejemploPila1 {
    public static void Main() {
        Stack miPila = new Stack();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IEnumerator miEnumerador = miPila.GetEnumerator();
        while (miEnumerador.MoveNext())
            Console.WriteLine("{0}", miEnumerador.Current);
    }
}
```

que escribiría

Contenido:

```
yo
soy
Hola,
```

Nota: los "enumeradores" existen también en otras plataformas, como Java, aunque allí reciben el nombre de "iteradores".

Se puede saber más sobre las estructuras dinámicas que hay disponibles en la plataforma .Net consultando la referencia en línea de MSDN (mucha de la cual está sin traducir al español):

[http://msdn.microsoft.com/es-es/library/system.collections\(en-us,VS.71\).aspx#](http://msdn.microsoft.com/es-es/library/system.collections(en-us,VS.71).aspx#)

8.7. Cómo "imitar" una pila usando "arrays"

Las estructuras dinámicas se pueden imitar usando estructuras estáticas sobredimensionadas, y esto puede ser un ejercicio de programación interesante. Por ejemplo, podríamos imitar una pila dando los siguientes pasos:

- Utilizamos internamente un array más grande que la cantidad de datos que esperemos que vaya a almacenar la pila.
- Creamos una función "Apilar", que añade en la primera posición libre del array (initialmente la 0) y después incrementa esa posición, para que el siguiente dato se introduzca a continuación.
- Creamos también una función "Desapilar", que devuelve el dato que hay en la última posición, y que disminuye el contador que indica la posición, de modo que el siguiente dato que se obtuviera sería el que se introdujo con anterioridad a éste.

El fuente podría ser así:

```
/*-----*/
/* Ejemplo en C#
/* pilaestatica.cs */
/*
/* Ejemplo de clase "Pila" */
/* basada en un array */
/*
/* Introducción a C#,
/* Nacho Cabanes
/*-----*/
```

```
using System;
using System.Collections;

public class PilaString {

    string[] datosPila;
    int posicionPila;
    const int MAXPILA = 200;
```

```

public static void Main() {
    string palabra;

    PilaString miPila = new PilaString();
    miPila.Apilar("Hola,");
    miPila.Apilar("soy");
    miPila.Apilar("yo");

    for (byte i=0; i<3; i++) {
        palabra = (string) miPila.Desapilar();
        Console.WriteLine( palabra );
    }
}

// Constructor
public PilaString() {
    posicionPila = 0;
    datosPila = new string[MAXPILA];
}

// Añadir a la pila: Apilar
public void Apilar(string nuevoDato) {
    if (posicionPila == MAXPILA)
        Console.WriteLine("Pila llena!");
    else {
        datosPila[posicionPila] = nuevoDato;
        posicionPila++;
    }
}

// Extraer de la pila: Desapilar
public string Desapilar() {
    if (posicionPila < 0)
        Console.WriteLine("Pila vacia!");
    else {
        posicionPila--;
        return datosPila[posicionPila];
    }
    return null;
}

} // Fin de la clase

```

Ejercicios propuestos:

- Usando esta misma estructura de programa, crear una clase "Cola", que permita introducir datos y obtenerlos en modo FIFO (el primer dato que se introduzca debe ser el primero que se obtenga). Debe tener un método "Encolar" y otro "Desencolar".
- Crear una clase "ListaOrdenada", que almacene un único dato (no un par clave-valor como los SortedList). Debe contener un método "Insertar", que añadirá un nuevo dato en orden en el array, y un "Extraer(n)", que obtenga un elemento de la lista (el número "n").

8.8. Los punteros en C#.

8.8.1 ¿Qué es un puntero?

La palabra "puntero" se usa para referirse a una **dirección de memoria**. Lo que tiene de especial es que normalmente un puntero tendrá un tipo de datos asociado: por ejemplo, un "puntero a entero" será una dirección de memoria en la que habrá almacenado (o podremos almacenar) un número entero.

El hecho de poder acceder directamente al contenido de ciertas posiciones de memoria da una mayor versatilidad a un programa, porque permite hacer casi cualquier cosa, pero a cambio de un riesgo de errores mucho mayor.

En lenguajes como C, es imprescindible utilizar punteros para poder crear estructuras dinámicas, pero en C# podemos "esquivarlos", dado que tenemos varias estructuras dinámicas ya creadas como parte de las bibliotecas auxiliares que acompañan al lenguaje básico. Aun así, veremos algún ejemplo que nos muestre qué es un puntero y cómo se utiliza.

En primer lugar, comentemos la sintaxis básica que utilizaremos:

```
int numero;          /* "numero" es un número entero */
int* posicion;      /* "posicion" es un "puntero a entero" (dirección de
                     memoria en la que podremos guardar un entero) */
```

Es decir, pondremos un asterisco entre el tipo de datos y el nombre de la variable. Ese asterisco puede ir junto a cualquiera de ambos, también es correcto escribir

```
int *posicion;
```

El valor que guarda "posicion" es una dirección de memoria. Generalmente no podremos hacer cosas como `posicion=5`; porque nada nos garantiza que la posición 5 de la memoria esté disponible para que nosotros la usemos. Será más habitual que tomemos una dirección de memoria que ya contiene otro dato, o bien que le pidamos al compilador que nos reserve un espacio de memoria (más adelante veremos cómo).

Si queremos que "posicion" contenga la dirección de memoria que el compilador había reservado para la variable "numero", lo haríamos usando el símbolo "&", así:

```
posicion = &numero;
```

8.8.2 Zonas "inseguras": unsafe

Como los punteros son "peligrosos" (es frecuente que den lugar a errores muy difíciles de encontrar), el compilador nos obligamos a que le digamos que sabemos que esa zona de programa no es segura, usando la palabra "unsafe":

```
unsafe static void pruebaPunteros() { ... }
```

Es más, si intentamos compilar obtendremos un mensaje de error, que nos dice que si nuestro código no es seguro, deberemos compilarlo con la opción "unsafe":

```
mcs unsafe1.cs
unsafe1.cs(15,31): error CS0227: Unsafe code requires the `unsafe` command line
option to be specified
Compilation failed: 1 error(s), 0 warnings
```

Por tanto, deberemos compilar con la opción /unsafe como forma de decir al compilador "sí, sé que este programa tiene zonas no seguras, pero aun así quiero compilarlo":

```
mcs unsafe1.cs /unsafe
```

8.8.3 Uso básico de punteros

Veamos un ejemplo básico de cómo dar valor a un puntero y de cómo guardar un valor en él:

```
/*
 * Ejemplo en C#
 */
/* unsafe1.cs
 */
/*
 */
/* Ejemplo de punteros (1)
 */
/*
 */
/* Introducción a C#, Nacho Cabanes
 */
/*
 */

using System;

public class ejemploUnsafe1 {

    private unsafe static void pruebaPunteros() {
        int* punteroAEntero;
        int x;

        // Damos un valor a x
        x = 2;
        // punteroAEntero será la dirección de memoria de x
        punteroAEntero = &x;
        // Los dos están en la misma dirección:
        Console.WriteLine("x vale {0}", x);
        Console.WriteLine("En punteroAEntero hay un {0}", *punteroAEntero);

        // Ahora cambiamos el valor guardado en punteroAEntero
        *punteroAEntero = 5;
        // Y x se modifica también:
        Console.WriteLine("x vale {0}", x);
        Console.WriteLine("En punteroAEntero hay un {0}", *punteroAEntero);
    }

    public static void Main() {
        pruebaPunteros();
    }
}
```

La salida de este programa es:

```
x vale 2
En punteroAEntero hay un 2
x vale 5
En punteroAEntero hay un 5
```

Es decir, cada cambio que hacemos en "x" se refleja en "punteroAEntero" y viceversa.

8.8.4 Zonas inseguras

También podemos hacer que sólo una parte de un programa sea insegura, indicando entre llaves una parte de una función:

```
/*-----*/
/* Ejemplo en C# */
/* unsafe2.cs */
/*
/* Ejemplo de punteros (2) */
/*
/* Introducción a C#, */
/* Nacho Cabanes */
/*-----*/

using System;

public class ejemploUnsafe2 {
    public unsafe static void Incrementar(int* p)
    {
        //Incrementamos el entero "apuntado" por p
        *p = *p + 1;
    }

    public static void Main()
    {
        int i = 1;

        // Ésta es la parte insegura de "Main"
        unsafe
        {
            // La función espera un puntero, así que le pasamos
            // la dirección de memoria del entero:
            Incrementar(&i);
        }

        // Y mostramos el resultado
        Console.WriteLine(i);
    }
}
```

8.8.4 Reservar espacio: stackalloc

Podemos reservar espacio para una variable dinámica usando "stackalloc". Por ejemplo, una forma alternativa de crear un array de enteros sería ésta:

```
int* datos = stackalloc int[5];
```

Un ejemplo completo podría ser:

```
/*-----*/
/* Ejemplo en C# */
/* unsafe3.cs */
/*
/* Ejemplo de punteros (3) */
/*
/* Introduccion a C#, */
/* Nacho Cabanes */
/*-----*/

using System;

public class EjemploUnsafe3 {
    public unsafe static void Main()
    {

        const int tamanyoArray = 5;
        int* datos = stackalloc int[tamanyoArray];

        // Rellenamos el array
        for (int i = 0; i < tamanyoArray; i++)
        {
            datos[i] = i*10;
        }

        // Mostramos el array
        for (int i = 0; i < tamanyoArray; i++)
        {
            Console.WriteLine(datos[i]);
        }
    }
}
```

Existen ciertas diferencias entre esta forma de trabajar y la que ya conocíamos: la memoria se reserva en la pila (stack), en vez de hacerlo en la zona de memoria "habitual", conocida como "heap" o montón, pero es un detalle sobre el que no vamos a profundizar.

8.8.5 Aritmética de punteros

Si aumentamos o disminuimos el valor de un puntero, cambiará la posición que representa... pero no cambiará de uno en uno, sino que saltará a la siguiente posición capaz de almacenar un dato como el que corresponde a su tipo base. Por ejemplo, si un puntero a entero tiene como valor 40.000 y hacemos "puntero++", su dirección pasará a ser 40.004 (porque cada entero ocupa 4 bytes). Vamos a verlo con un ejemplo:

```

/*
/* Ejemplo en C#
/* unsafe4.cs
/*
/* Ejemplo de punteros (4)
/*
/* Introduccion a C#,
/* Nacho Cabanes
/*
using System;

public class EjemploUnsafe4 {
    public unsafe static void Main()
    {
        const int tamanyoArray = 5;
        int* datos = stackalloc int[tamanyoArray];
        int* posicion = datos;

        Console.WriteLine("Posicion actual: {0}", (int) posicion);

        // Ponemos un 0 en el primer dato
        *datos = 0;

        // Rellenamos los demás con 10,20,30...
        for (int i = 1; i < tamanyoArray; i++)
        {
            posicion++;
            Console.WriteLine("Posicion actual: {0}", (int) posicion);
            *posicion = i * 10;
        }

        // Finalmente mostramos el array
        Console.WriteLine("Contenido:");
        for (int i = 0; i < tamanyoArray; i++)
        {
            Console.WriteLine(datos[i]);
        }
    }
}

```

El resultado sería algo parecido (porque las direcciones de memoria que obtengamos no tienen por qué ser las mismas) a:

```

Posicion actual: 1242196
Posicion actual: 1242200
Posicion actual: 1242204
Posicion actual: 1242208
Posicion actual: 1242212
Contenido:
0

```

```
10
20
30
40
```

8.8.6 La palabra "fixed"

C# cuenta con un "recolector de basura", que se encarga de liberar el espacio ocupado por variables que ya no se usan. Esto puede suponer algún problema cuando usamos variables dinámicas, porque estemos accediendo a una posición de memoria que el entorno de ejecución haya previsto que ya no necesitaríamos... y haya borrado.

Por eso, en ciertas ocasiones el compilador puede avisarnos de que algunas asignaciones son peligrosas y obligar a que usemos la palabra "fixed" para indicar al compilador que esa zona "no debe limpiarse automáticamente".

Esta palabra se usa antes de la declaración y asignación de la variable, y la zona de programa que queremos "bloquear" se indica entre llaves:

```
/*-----*/
/* Ejemplo en C#           */
/* unsafe5.cs               */
/*                         */
/* Ejemplo de punteros (5) */
/*                         */
/* Introduccion a C#,      */
/* Nacho Cabanes           */
/*-----*/
using System;

public class EjemploUnsafe5 {
    public unsafe static void Main()
    {
        int[] datos={10,20,30};

        Console.WriteLine("Leyendo el segundo dato...");
        fixed (int* posicionDatos = &datos[1])
        {
            Console.WriteLine("En posicionDatos hay {0}", *posicionDatos);
        }

        Console.WriteLine("Leyendo el primer dato...");
        fixed (int* posicionDatos = datos)
        {
            Console.WriteLine("Ahora en posicionDatos hay {0}", *posicionDatos);
        }
    }
}
```

Como se ve en el programa anterior, en una zona "fixed" no se puede modificar el valor de esa variables; si lo intentamos recibiremos un mensaje de error que nos avisa de que esa variable es de "sólo lectura" (read-only). Por eso, para cambiarla, tendremos que empezar otra nueva zona "fixed".

El resultado del ejemplo anterior sería:

```
Leyendo el segundo dato...
En posicionDato hay 20
Leyendo el primer dato...
Ahora en posicionDato hay 10
```

9. Otras características avanzadas de C#

9.1. Espacios de nombres

Desde nuestros primeros programas hemos estado usando cosas como "System.Console" o bien "using System". Esa palabra "System" indica que las funciones que estamos usando pertenecen a la estructura básica de C# y de la plataforma .Net.

La idea detrás de ese "using" es que puede ocurrir que distintos programadores en distintos puntos del mundo crean funciones o clases que se llamen igual, y, si se mezclan fuentes de distintas procedencias, esto podría dar lugar a programas que no compilaran correctamente, o, peor aún, que compilaran pero no funcionaran de la forma esperada.

Por eso, se recomienda usar "espacios de nombres", que permitan distinguir unos de otros. Por ejemplo, si yo quisiera crear mi propia clase "Console" para el acceso a la consola, o mi propia clase "Random" para manejo de números aleatorios, lo razonable es crear un nuevo espacio de nombres.

De hecho, con entornos como SharpDevelop o Visual Studio, cuando creamos un nuevo proyecto, el fuente "casi vacío" que se nos propone contendrá un espacio de nombres que se llamará igual que el proyecto. Esta es apariencia del fuente si usamos VisualStudio 2008:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Y esta es apariencia del fuente si usamos SharpDevelop 3:

```
using System;

namespace PruebaDeNamespaces
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }

        // TODO: Implement Functionality Here
    }
}
```

```

        Console.WriteLine("Press any key to continue . . . ");
        Console.ReadKey(true);
    }
}

```

Vamos a un ejemplo algo más avanzado, que contenga un espacio de nombres, que cree una nueva clase Console y que utilice las dos (la nuestra y la original, de System):

```

/*
 * Ejemplo en C#
 */
/* namespaces.cs */
/*
 * Ejemplo de espacios de */
/* nombres */
/*
 * Introducción a C#, */
/* Nacho Cabanes */
/*
 */

using System;

namespace ConsolaDeNacho {
    public class Console
    {
        public static void WriteLine(string texto)
        {
            System.Console.ForegroundColor = ConsoleColor.Blue;
            System.Console.WriteLine("Mensaje: "+texto);
        }
    }
}

public class PruebaDeNamespaces
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
        ConsolaDeNacho.Console.WriteLine("Hola otra vez");
    }
}

```

Como se puede ver, este ejemplo tiene dos clases Console, y ambas tienen un método WriteLine. Una es la original de C#, que invocaríamos con "System.Console". Otra es la que hemos creado para el ejemplo, que escribe un texto modifica y en color (ayudándose de System.Console), y que llamaríamos mediante "ConsolaDeNacho.Console". El resultado es que podemos tener dos clases Console accesibles desde el mismo programa, sin que existan conflictos entre ellas. El resultado del programa sería:

```

Hola
Mensaje: Hola otra vez

```

9.2. Operaciones con bits

Podemos hacer desde C# operaciones entre bits de dos números (AND, OR, XOR, etc). Vamos primero a ver qué significa cada una de esas operaciones.

Operación	Resultado	En C#	Ejemplo
Complemento (not)	Cambiar 0 por 1 y viceversa	\sim	$\sim 1100 = 0011$
Producto lógico (and)	1 sólo si los 2 bits son 1	&	$1101 \& 1011 = 1001$
Suma lógica (or)	1 sólo si uno de los bits es 1		$1101 1011 = 1001$
Suma exclusiva (xor)	1 sólo si los 2 bits son distintos	\wedge	$1101 \wedge 1011 = 0110$
Desplazamiento a la izquierda	Desplaza y rellena con ceros	$<<$	$1101 << 2 = 110100$
Desplazamiento a la derecha	Desplaza y rellena con ceros	$>>$	$1101 >> 2 = 0011$

Ahora vamos a aplicarlo a un ejemplo completo en C#:

```
/*
 * Ejemplo en C#
 */
/* bits.cs */
/*
 */
/* Operaciones entre bits */
/*
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes
*/
/*
 */

using System;

public class bits
{
    public static void Main()
    {
        int a    = 67;
        int b    = 33;

        Console.WriteLine("La variable a vale {0}", a);
        Console.WriteLine("y b vale {0}", b);
        Console.WriteLine("El complemento de a es: {0}", ~a);
        Console.WriteLine("El producto lógico de a y b es: {0}", a&b);
        Console.WriteLine("Su suma lógica es: {0}", a|b);
        Console.WriteLine("Su suma lógica exclusiva es: {0}", a^b);
        Console.WriteLine("Desplacemos a a la izquierda: {0}", a << 1);
        Console.WriteLine("Desplacemos a a la derecha: {0}", a >> 1);

    }
}
```

El resultado es:

```
La variable a vale 67
y b vale 33
El complemento de a es: -68
El producto lógico de a y b es: 1
Su suma lógica es: 99
Su suma lógica exclusiva es: 98
```

Desplazemos a a la izquierda: 134
 Desplazemos a a la derecha: 33

Para comprobar que es correcto, podemos convertir al sistema binario esos dos números y seguir las operaciones paso a paso:

```
67 = 0100 0011
33 = 0010 0001
```

En primer lugar complementamos "a", cambiando los ceros por unos:

```
1011 1100 = -68
```

Después hacemos el producto lógico de A y B, multiplicando cada bit, de modo que $1*1 = 1$, $1*0 = 0$, $0*0 = 0$

```
0000 0001 = 1
```

Después hacemos su suma lógica, sumando cada bit, de modo que $1+1 = 1$, $1+0 = 1$, $0+0 = 0$

```
0110 0011 = 99
```

La suma lógica exclusiva devuelve un 1 cuando los dos bits son distintos: $1\wedge1 = 0$, $1\wedge0 = 1$, $0\wedge0 = 0$

```
0110 0010 = 98
```

Desplazar los bits una posición a la izquierda es como multiplicar por dos:

```
1000 0110 = 134
```

Desplazar los bits una posición a la derecha es como dividir entre dos:

```
0010 0001 = 33
```

¿Qué utilidades puede tener todo esto? Posiblemente, más de las que parece a primera vista. Por ejemplo: desplazar a la izquierda es una forma muy rápida de multiplicar por potencias de dos; desplazar a la derecha es dividir por potencias de dos; la suma lógica exclusiva (xor) es un método rápido y sencillo de cifrar mensajes; el producto lógico nos permite obligar a que ciertos bits sean 0 (algo que se puede usar para comprobar máscaras de red); la suma lógica, por el contrario, puede servir para obligar a que ciertos bits sean 1...

Un último comentario: igual que hacíamos operaciones abreviadas como

```
x += 2;
```

también podremos hacer cosas como

```
x <= 2;
x &= 2;
x |= 2;
...
```

9.3. Enumeraciones

Cuando tenemos varias constantes, cuyos valores son números enteros, hasta ahora estamos dando los valores uno por uno, así:

```
const int LUNES = 0, MARTES = 1,
MIERCOLES = 2, JUEVES = 3,
VIERNES = 4, SABADO = 5,
DOMINGO = 6;
```

Hay una forma alternativa de hacerlo, especialmente útil si son números enteros consecutivos. Se trata de **enumerarlos**:

```
enum diasSemana { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO,
DOMINGO };
```

(Al igual que las constantes de cualquier otro tipo, se puede escribir en mayúsculas para recordar "de un vistazo" que son constantes, no variables)

La primera constante valdrá 0, y las demás irán aumentando de una en una, de modo que en nuestro caso valen:

```
LUNES = 0, MARTES = 1, MIERCOLES = 2, JUEVES = 3, VIERNES = 4,
SABADO = 5, DOMINGO = 6
```

Si queremos que los valores no sean exactamente estos, podemos dar valor a cualquiera de las constantes, y las siguientes irán aumentando de uno en uno. Por ejemplo, si escribimos

```
enum diasSemana { LUNES=1, MARTES, MIERCOLES, JUEVES=6, VIERNES,
SABADO=10, DOMINGO };
```

Ahora sus valores son:

```
LUNES = 1, MARTES = 2, MIERCOLES = 3, JUEVES = 6, VIERNES = 7,
SABADO = 10, DOMINGO = 11
```

Un ejemplo básico podría ser

```
/*
-----*/
/* Ejemplo en C# */
/* enum.cs */
/*
*/
/* Ejemplo de enumeraciones */
/*
*/
/* Introduccion a C#,
/* Nacho Cabanes */
/*
-----*/

using System;

public class enumeraciones
{

    enum diasSemana { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO,
DOMINGO };

    public static void Main()
    {

        Console.WriteLine("En la enumeracion, el miércoles tiene el valor: {0} ",
```

```

    diasSemana.MIERCOLES);
Console.WriteLine("que equivale a: {0}",
    (int) diasSemana.MIERCOLES);

const int LUNES = 0, MARTES = 1, MIERCOLES = 2, JUEVES = 3,
VIERNES = 4, SABADO = 5, DOMINGO = 6;

Console.WriteLine("En las constantes, el miércoles tiene el valor: {0}",
    MIERCOLES);

}
}

```

y su resultado será:

```

En la enumeracion, el miércoles tiene el valor: MIERCOLES que equivale a: 2
En las constantes, el miércoles tiene el valor: 2

```

Nosotros hemos usado enumeraciones muchas veces hasta ahora, sin saber realmente que lo estábamos haciendo. Por ejemplo, el modo de apertura de un fichero (FileMode) es una enumeración, por lo que escribimos FileMode.Open. También son enumeraciones los códigos de color de la consola (como ConsoleColor.Red) y las teclas de la consola (como ConsoleKey.Escape).

Nota: las enumeraciones existen también en otros lenguajes como C y C++, pero la sintaxis es ligeramente distinta: en C# es necesario indicar el nombre de la enumeración cada vez que se usen sus valores (como en diasSemana.MIERCOLES), mientras que en C se usa sólo el valor (MIERCOLES).

9.4. Propiedades

Hasta ahora estábamos siguiendo la política de que los atributos de una clase sean privados, y se acceda a ellos a través de métodos "get" (para leer su valor) y "set" (para cambiarlo). En el caso de C#, existe una forma alternativa de conseguir el mismo efecto, empleando las llamadas "propiedades", que tienen una forma abreviada de escribir sus métodos "get" y "set":

```

/*
 *----- Ejemplo en C#
 *----- propiedades.cs
 *----- Ejemplo de propiedades
 *----- Introduccion a C#,
 *----- Nacho Cabanes
 *----- */
using System;

public class EjemploPropiedades
{
    // -----
    // Un atributo convencional, privado

```

```
private int altura = 0;

// Para ocultar detalles, leemos su valor con un "get"
public int GetAltura()
{
    return altura;
}

// Y lo fijamos con un "set"
public void SetAltura(int nuevoValor)
{
    altura = nuevoValor;
}

// -----
// Otro atributo convencional, privado
private int anchura = 0;

// Oculto mediante una "propiedad"
public int Anchura
{
    get
    {
        return anchura;
    }

    set
    {
        anchura = value;
    }
}

// -----
// El "Main" de prueba
public static void Main()
{
    EjemploPropiedades ejemplo
        = new EjemploPropiedades();

    ejemplo.SetAltura(5);
    Console.WriteLine("La altura es {0}",
        ejemplo.GetAltura());

    ejemplo.Anchura = 6;
    Console.WriteLine("La anchura es {0}",
        ejemplo.Anchura);
}
```

Al igual que ocurría con las enumeraciones, ya hemos usado "propiedades" anteriormente, sin saberlo: la longitud ("Length") de una cadena, el tamaño ("Length") y la posición actual ("Position") en un fichero, el título ("Title") de una ventana en consola, etc.

Una curiosidad: si una propiedad tiene un "get", pero no un "set", será una propiedad de sólo lectura, no podremos hacer cosas como "Anchura = 4", porque el programa no compilaría. De igual modo, se podría crear una propiedad de sólo escritura, definiendo su "set" pero no su "get".

9.5. Parámetros de salida (out)

Hemos hablado de dos tipos de parámetros de una función: parámetros por valor (que no se pueden modificar) y parámetros por referencia ("ref", que sí se pueden modificar). Un uso habitual de los parámetros por referencia es devolver más de un valor a la salida de una función. Para ese uso, en C# existe otra alternativa: los parámetros de salida. Se indican con la palabra "out" en vez de "ref", y no exigen que las variables tengan un valor inicial:

```
public void ResolverEcuacionSegundoGrado(  
    float a, float b, float c,  
    out float x1, out float x2)
```

9.6. Introducción a las expresiones regulares.

Las "expresiones regulares" permiten hacer comparaciones mucho más abstractas que si se usa un simple "IndexOf". Por ejemplo, podemos comprobar con una orden breve si todos los caracteres de una cadena son numéricos, o si empieza por mayúscula y el resto son minúsculas, etc.

Vamos a ver solamente un ejemplo con un caso habitual: comprobar si una cadena es numérica, alfabética o alfanumérica. Las ideas básicas son:

- Usaremos el tipo de datos "RegEx" (expresión regular).
- Tenemos un método `IsMatch`, que devuelve "true" si una cadena de texto coincide con un cierto patrón.
- Uno de los patrones más habituales es indicar un rango de datos: [a-z] quiere decir "un carácter entre la a y la z".
- Podemos añadir modificadores: * para indicar "0 o más veces", + para "1 o más veces", ? para "0 o una vez", como en [a-z]+, que quiere decir "uno o más caracteres entre la a y la z".
- Aun así, esa expresión puede dar resultados inesperados: un secuencia como [0-9]+ aceptaría cualquier cadena que contuviera una secuencia de números... aunque tuviera otros símbolos al principio y al final. Por eso, si queremos que sólo tenga cifras numéricas, nuestra expresión regular debería ser "inicio de cadena, cualquier secuencia de cifras, final de cadena", que se representaría como "\A[0-9]+\z". Una alternativa es plantear la expresión regular al contrario: "no es válido si contiene algo que no sea del 0 al 9", que se podría conseguir devolviendo lo contrario de lo que indique la expresión "[^0-9]".

El ejemplo podría ser:

```

using System;
using System.Text.RegularExpressions;

class PruebaExprRegulares
{
    public static bool EsNumeroEntero(String cadena)
    {
        Regex patronNumerico = new Regex("[^0-9]");
        return !patronNumerico.IsMatch(cadena);
    }

    public static bool EsNumeroEntero2(String cadena)
    {
        Regex patronNumerico = new Regex(@"\A[0-9]*\z");
        return patronNumerico.IsMatch(cadena);
    }

    public static bool EsNumeroConDecimales(String cadena)
    {
        Regex patronNumericoConDecimales =
            new Regex(@"\A[0-9]*,[0-9]+\z");
        return patronNumericoConDecimales.IsMatch(cadena);
    }

    public static bool EsAlfabetico(String cadena)
    {
        Regex patronAlfabetico = new Regex("[^a-zA-Z]");
        return !patronAlfabetico.IsMatch(cadena);
    }

    public static bool EsAlfanumerico(String cadena)
    {
        Regex patronAlfanumerico = new Regex("[^a-zA-Z0-9]");
        return !patronAlfanumerico.IsMatch(cadena);
    }

    static void Main(string[] args)
    {
        if (EsNumeroEntero("hola"))
            Console.WriteLine("hola es número entero");
        else
            Console.WriteLine("hola NO es número entero");

        if (EsNumeroEntero("1942"))
            Console.WriteLine("1942 es un número entero");
        else
            Console.WriteLine("1942 NO es un número entero");

        if (EsNumeroEntero2("1942"))
            Console.WriteLine("1942 es entero (forma 2)");
        else
            Console.WriteLine("1942 NO es entero (forma 2)");
    }
}

```

```

if (EsNumeroEntero("23,45"))
    Console.WriteLine("23,45 es un número entero");
else
    Console.WriteLine("23,45 NO es un número entero");

if (EsNumeroEntero2("23,45"))
    Console.WriteLine("23,45 es entero (forma 2)");
else
    Console.WriteLine("23,45 NO es entero (forma 2)");

if (EsNumeroConDecimales("23,45"))
    Console.WriteLine("23,45 es un número con decimales");
else
    Console.WriteLine("23,45 NO es un número con decimales");

if (EsNumeroConDecimales("23,45,67"))
    Console.WriteLine("23,45,67 es un número con decimales");
else
    Console.WriteLine("23,45,67 NO es un número con decimales");

if (EsAlfabetico("hola"))
    Console.WriteLine("hola es alfabetico");
else
    Console.WriteLine("hola NO es alfabetico");

if (EsAlfanumerico("hola1"))
    Console.WriteLine("hola1 es alfanumerico");
else
    Console.WriteLine("hola1 NO es alfanumerico");
}
}

```

Su salida es:

```

hola NO es número entero
1942 es un número entero
1942 es entero (forma 2)
23,45 NO es un número entero
23,45 NO es entero (forma 2)
23,45 es un número con decimales
23,45,67 NO es un número con decimales
hola es alfabetico
hola1 es alfanumerico

```

Las expresiones regulares son algo complejo. Por una parte, su sintaxis puede llegar a ser difícil de seguir. Por otra parte, el manejo en C# no se limita a buscar, sino que también permite otras operaciones, como reemplazar unas expresiones por otras. Como ver muchos más detalles podría hacer el texto demasiado extenso, puede ser recomendable ampliar información usando la página web de MSDN (Microsoft Developer Network):

[http://msdn.microsoft.com/es-es/library/system.text.regularexpressions.regex\(VS.80\).aspx](http://msdn.microsoft.com/es-es/library/system.text.regularexpressions.regex(VS.80).aspx)

9.7. El operador coma

Cuando vimos la orden "for", siempre usábamos una única variable como contador, pero esto no tiene por qué ser siempre así. Vamos a verlo con un ejemplo:

```
/*
/*----- Ejemplo en C#
/* coma.cs
/*
/*
/* Operador coma
/*
/*
/* Introduccion a C#, Nacho Cabanes
/*
/*-----*/
using System;

public class coma
{
    public static void Main()
    {
        int i, j;

        for (i=0, j=1; i<=5 && j<=30; i++, j+=2)
            Console.WriteLine("i vale {0} y j vale {0}.", i, j);
    }
}
```

Vamos a ver qué hace este "for":

- Los valores iniciales son $i=0, j=1$.
- Se repetirá mientras que $i \leq 5$ y $j \leq 30$.
- Al final de cada paso, i aumenta en una unidad, y j en dos unidades.

El resultado de este programa es:

```
i vale 0 y j vale 0.
i vale 1 y j vale 1.
i vale 2 y j vale 2.
i vale 3 y j vale 3.
i vale 4 y j vale 4.
i vale 5 y j vale 5.
```

Nota: En el lenguaje C se puede "rizar el rizo" todavía un poco más: la condición de terminación también podría tener una coma, y entonces no se sale del bucle "for" hasta que se cumplen las dos condiciones (algo que no es válido en C#, ya que la condición debe ser un "Boolean", y la coma no es un operador válido para operaciones booleanas):

```
for (i=0, j=1; i<=5, j<=30; i++, j+=2)
```

9.8. Lo que no vamos a ver...

En C# hay más que lo que hemos visto aquí. Mencionaremos algunos, por si alguien quiere ampliar información por su cuenta en MSDN o en cualquier otra fuente de información. Por ejemplo:

- Delegados (delegate).
- Indexadores.
- Colecciones genéricas.
- ...

10. Algunas bibliotecas adicionales de uso frecuente

10.1. Más posibilidades de la "consola"

En "Console" hay mucho más que ReadLine y WriteLine, aunque quizás no todas las posibilidades estén contempladas en implementaciones "alternativas", como las primeras versiones de Mono. Vamos a ver algunas de las posibilidades que nos pueden resultar más útiles:

- Clear: borra la pantalla.
- ForegroundColor: cambia el color de primer plano (para indicar los colores, hay definidas constantes como "ConsoleColor.Black", que se detallan al final de este apartado).
- BackgroundColor: cambia el color de fondo (para el texto que se escriba a partir de entonces; si se quiere borrar la pantalla con un cierto color, se deberá primero cambiar el color de fondo y después usar "Clear").
- SetCursorPosition(x, y): cambia la posición del cursor ("x" se empieza a contar desde el margen izquierdo, e "y" desde la parte superior de la pantalla).
- Readkey(interceptar): lee una tecla desde teclado. El parámetro "interceptar" es un "bool", y es opcional. Indica si se debe capturar la tecla sin permitir que se vea en pantalla ("true" para que no se vea, "false" para que se pueda ver). Si no se indica este parámetro, la tecla se muestra en pantalla.
- KeyAvailable: indica si hay alguna tecla disponible para ser leída (es un "bool")
- Title: el título que se va a mostrar en la consola (es un "string")

```
/*-----*/
/* Ejemplo en C# */
/* consola.cs */
/*
/* Más posibilidades de */
/* "System.Console" */
/*
/* Introducción a C#, */
/* Nacho Cabanes */
/*-----*/
using System;

public class consola
{
    public static void Main()
    {
        int posX, posY;

        Console.Title = "Ejemplo de consola";
        Console.BackgroundColor = ConsoleColor.Green;
        Console.ForegroundColor = ConsoleColor.Black;
        Console.Clear();

        posY = 10; // En la fila 10
```

```

Random r = new Random(DateTime.Now.Millisecond);
posX = r.Next(20, 40); // Columna al azar entre 20 y 40
ConsoleCursorPosition(posX, posY);
Console.WriteLine("Bienvenido");

Console.ForegroundColor = ConsoleColor.Blue;
ConsoleCursorPosition(10, 15);
Console.Write("Pulsa 1 o 2: ");
ConsoleKeyInfo tecla;
do
{
    tecla = Console.ReadKey(false);
}
while ((tecla.KeyChar != '1') && (tecla.KeyChar != '2'));

int maxY = Console.WindowHeight;
int maxX = Console.WindowWidth;
ConsoleCursorPosition(maxX-50, maxY-1);
Console.ForegroundColor = ConsoleColor.Red;
Console.Write("Pulsa una tecla para terminar... ");
Console.ReadKey(true);

}
}

```

(Nota: si se prueba este fuente desde Mono, habrá que compilar con "gmcs" en vez de con "mcs", para compilar usando la versión 2.x de la plataforma .Net, no la 1.x, que no tenía estas posibilidades).

Para comprobar el valor de una tecla, como se ve en el ejemplo anterior, tenemos que usar una variable de tipo "ConsoleKeyInfo" (información de tecla de consola). Un ConsoleKeyInfo tiene campos como:

- KeyChar, que representa el carácter que se escribiría al pulsar esa tecla. Por ejemplo, podríamos hacer `if (tecla.KeyChar == '1') ...`
- Key, que se refiere a la tecla (porque hay teclas que no tienen un carácter visualizable, como F1 o las teclas de cursor). Por ejemplo, para comprobar la tecla ESC podríamos hacer `if (tecla.Key == ConsoleKey.Escape) ...`. Algunos de los códigos de tecla disponibles son:
 - Teclas de edición y control como, como: Backspace (Tecla RETROCESO), Tab (Tecla TAB), Clear (Tecla BORRAR), Enter (Tecla ENTRAR), Pause (Tecla PAUSA), Escape (Tecla ESC (ESCAPE)), Spacebar (Tecla BARRA ESPACIADORA), PrintScreen (Tecla IMPR PANT), Insert (Tecla INS (INSERT)), Delete (Tecla SUPR (SUPRIMIR))
 - Teclas de movimiento del cursor, como: PageUp (Tecla RE PÁG), PageDown (Tecla AV PÁG), End (Tecla FIN), Home (Tecla INICIO), LeftArrow (Tecla FLECHA IZQUIERDA), UpArrow (Tecla FLECHA ARRIBA), RightArrow (Tecla FLECHA DERECHA), DownArrow (Tecla FLECHA ABAJO)
 - Teclas alfabéticas, como: A (Tecla A), B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
 - Teclas numéricas, como: D0 (Tecla 0), D1, D2, D3, D4, D5, D6, D7, D8, D9

- Teclado numérico adicional: NumPad0 (Tecla 0 del teclado numérico), NumPad1, NumPad2, NumPad3, NumPad4, NumPad5, NumPad6, NumPad7, NumPad8, NumPad9, Multiply (Tecla Multiplicación), Add (Tecla Agregar), Separator (Tecla Separador), Subtract (Tecla Resta), Decimal (Tecla Decimal), Divide (Tecla División)
 - Sleep (Tecla Espera del equipo)
 - Teclas de función: F1, F2 y sucesivas (hasta F24)
 - Teclas especiales de Windows: LeftWindows (Tecla izquierda con el logotipo de Windows), RightWindows (Tecla derecha con el logotipo de Windows)
 - Incluso teclas multimedia, si el teclado las incorpora, como: VolumeMute (Tecla Silenciar el volumen, en Microsoft Natural Keyboard, bajo Windows 2000 o posterior), VolumeDown (Bajar el volumen, ídem), VolumeUp (Subir el volumen), MediaNext (Tecla Siguiente pista de multimedia), etc.
 - Modifiers, que permite comprobar si se han pulsado simultáneamente teclas modificadoras: Alt, Shift o Control. Un ejemplo de su uso sería:
- ```
if ((tecla.Modifiers & ConsoleModifiers.Alt) != 0)
 Console.WriteLine("Has pulsado Alt");
```

Los colores que tenemos disponibles (y que se deben escribir precedidos con "ConsoleColor") son: Black (negro), DarkBlue (azul marino), DarkGreen (verde oscuro) DarkCyan (verde azulado oscuro), DarkRed (rojo oscuro), DarkMagenta (fucsia oscuro o púrpura), DarkYellow (amarillo oscuro u ocre), Gray (gris), DarkGray (gris oscuro), Blue (azul), Green (verde), Cyan (aguamarina o verde azulado claro), Red (rojo), Magenta (fucsia), Yellow (amarillo), White (blanco).

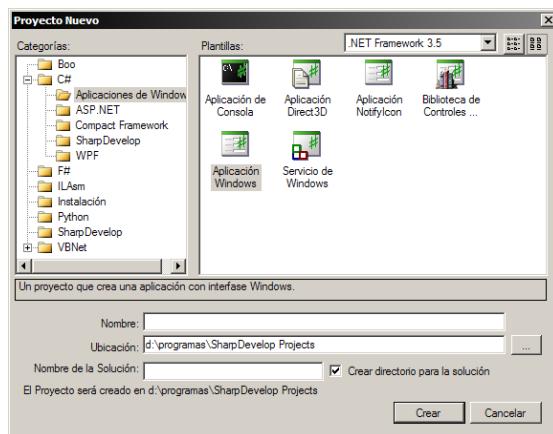
## 10.2. Nociónes básicas de entornos gráficos

En C# podemos crear con una cierta facilidad programas en entornos gráficos, con menús botones, listas desplegables, etc.

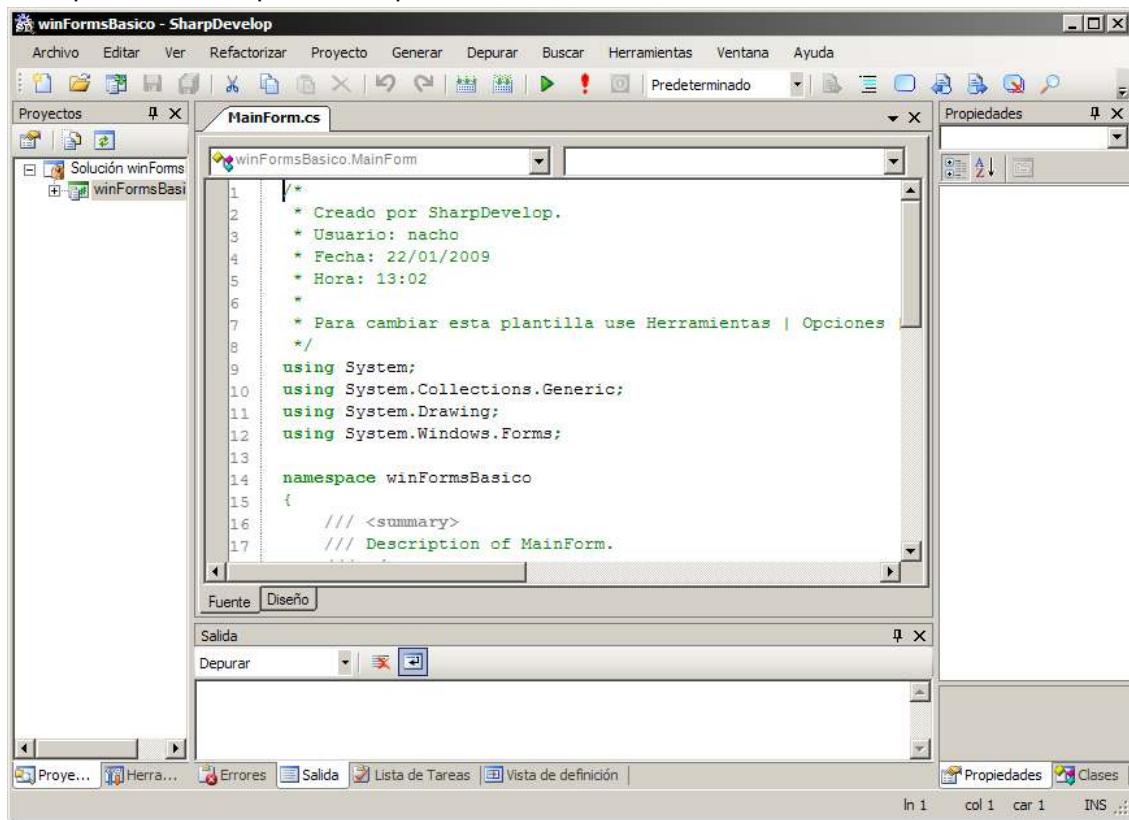
La forma más cómoda de conseguirlo es usando herramientas que incluyan un editor visual, como Visual Studio o SharpDevelop.

A pesar de que existen versiones gratuitas de Visual Studio, vamos a ver el caso de SharpDevelop, que necesita un ordenador menos potente y tiene un manejo muy similar (en el Apéndice 4 tienes los cambios para Visual Studio).

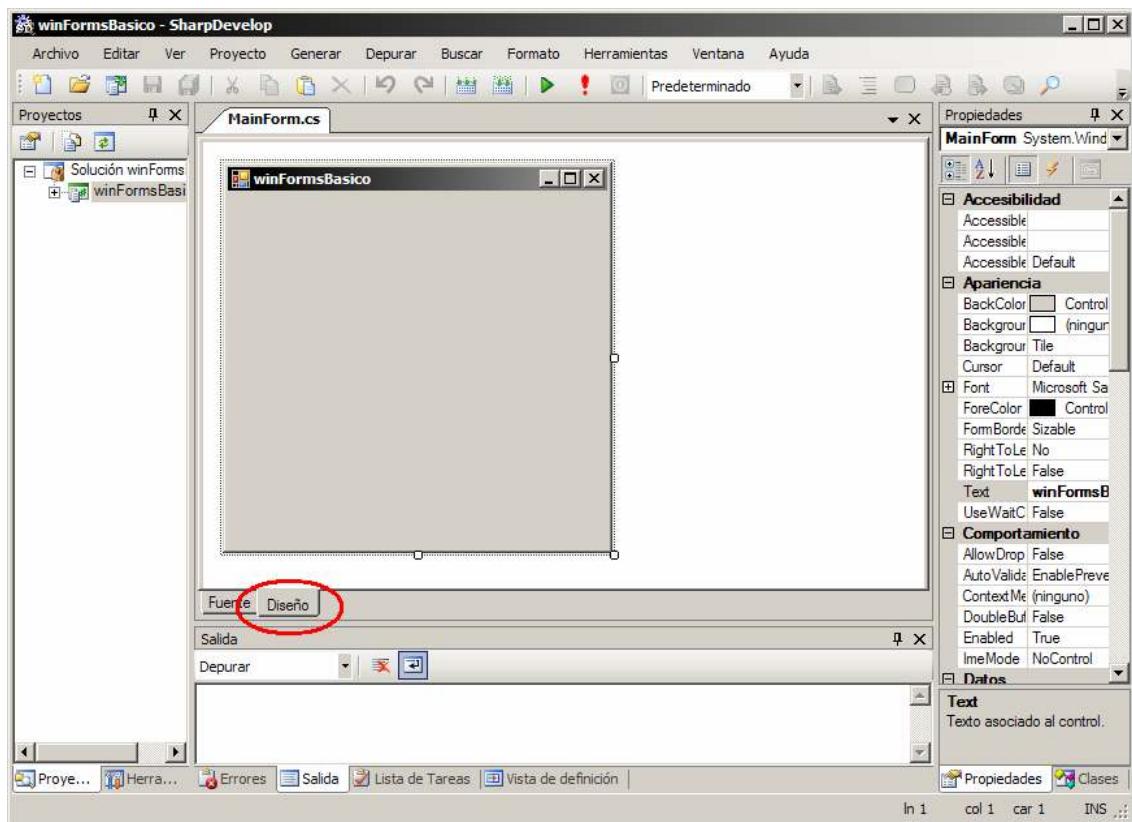
Cuando entramos a SharpDevelop, dirímos que queremos crear una "nueva solución", y en el menú escogeríamos "Aplicación Windows":



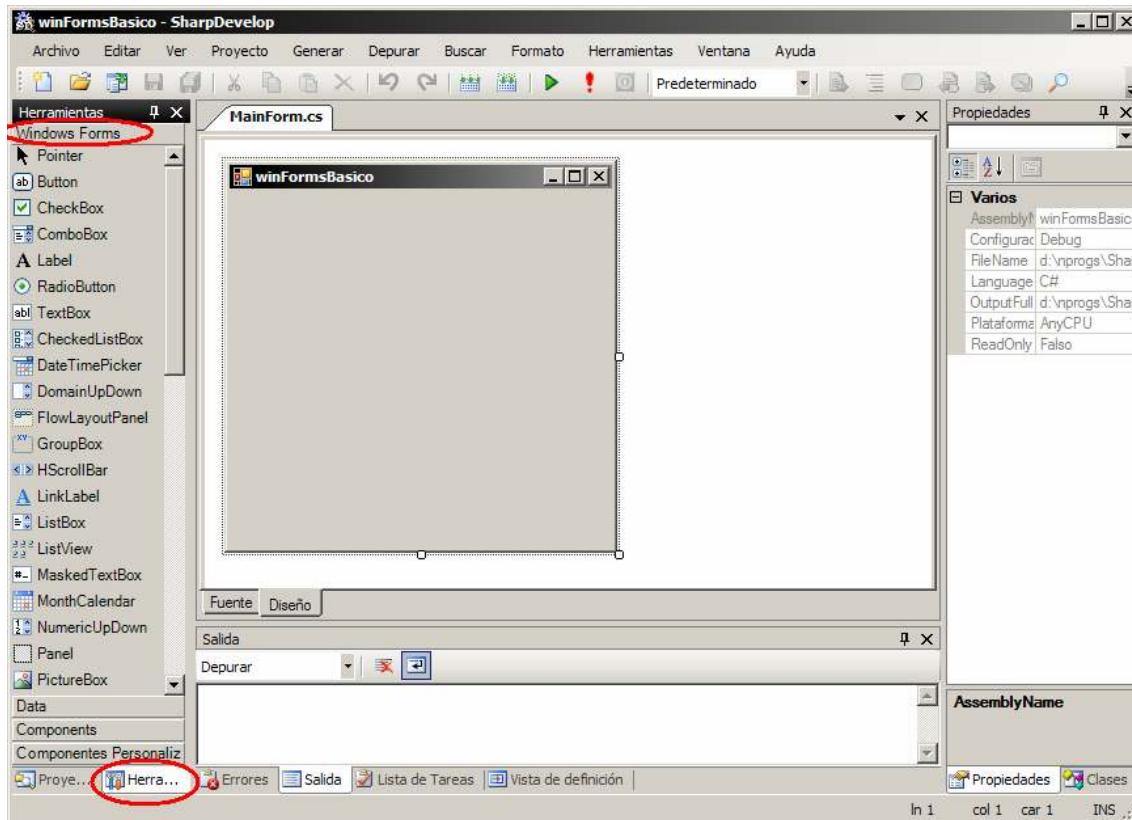
Nos aparecerá un esqueleto de aplicación:



Debajo de la ventana de código hay una pestaña llamada "Diseño", que nos permite acceder al diseñador visual:



Para poder incluir botones y otros elementos visuales, debemos escoger la ventana de "Herramientas" en la parte inferior de la pantalla, y luego el panel "Windows Forms" en esta ventana:

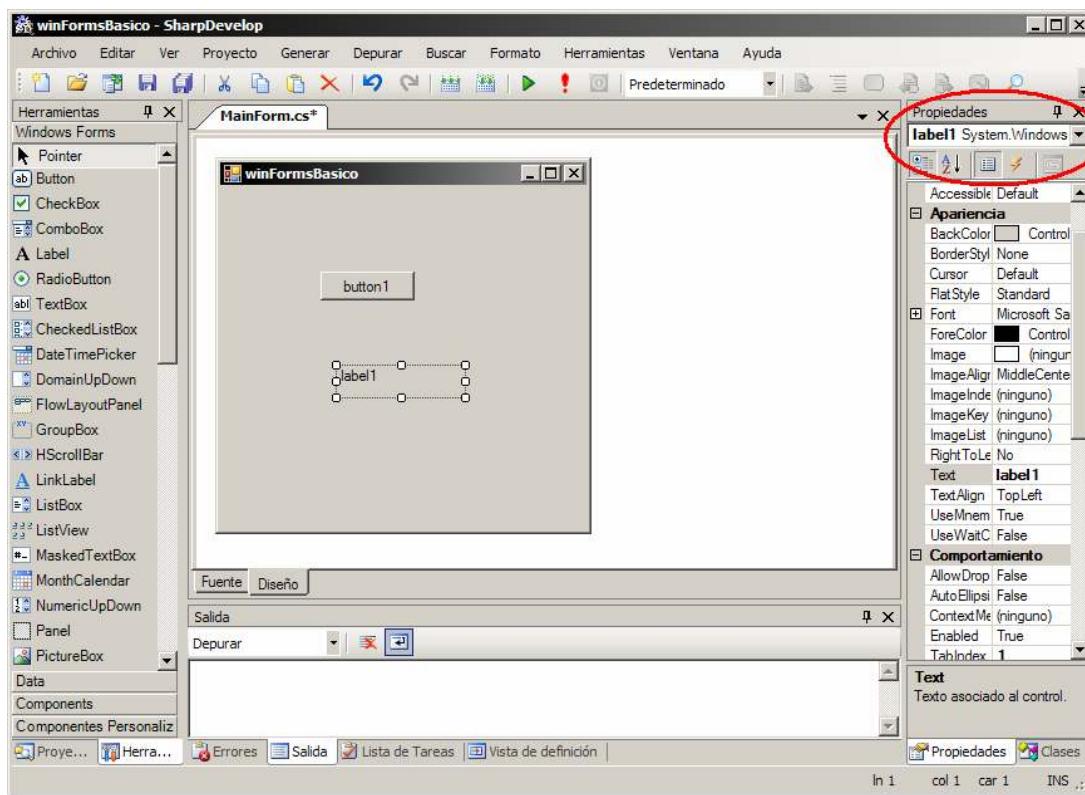


Para incluir un botón, podemos hacer clic en el elemento "Button" del panel izquierdo, y luego hacer un clic en la parte de la ventana en la que queremos que aparezca. De igual modo, podríamos añadir otros elementos.

Por ejemplo, vamos a añadir un botón (Button) y una etiqueta de texto (Label), para hacer un primer programa que cambie el texto de la etiqueta cuando pulsemos el botón.

Las propiedades de cada uno de estos elementos aparecen en la parte derecha, en una nueva ventana. Estas propiedades las podremos cambiar directamente en ese panel, o bien desde código. Algunas de esas propiedades son:

- Name, el nombre con el que se accederá desde el código.
- Text, el texto que muestra un elemento.
- ForeColor, el color con el que se muestra.
- TextAlign, para indicar la alineación del texto (y poder centrarlo, por ejemplo).
- Enabled, para poder activar o desactivar un elemento.
- Location, la posición en que se encuentra (que podemos ajustar inicialmente con el ratón).
- Size, el tamaño (ancho y alto, que también se puede ajustar inicialmente con el ratón).



Si queremos que al pulsar el botón cambie el texto, tendremos que modificar el código que corresponde al "evento" de pulsación del botón. Lo conseguimos simplemente haciendo doble clic en el botón, y aparece este texto:

```
void Button1Click(object sender, EventArgs e)
{
```

}

Dentro de ese método escribiremos lo que queremos que ocurra al hacer clic en el botón. Por ejemplo, para que el texto de la etiqueta "label1" pase a ser "Hola", haríamos:

```
void Button1Click(object sender, EventArgs e)
{
 label1.Text = "Hola";
}
```

#### Ejercicios propuestos:

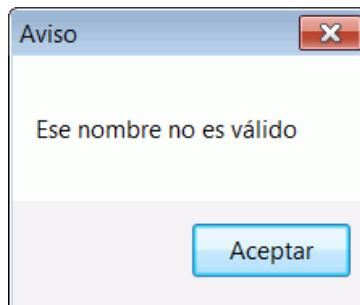
- Un programa que muestre una ventana con 3 recuadros de texto (TextBox) y un botón "Calcular suma". Cuando se pulse el botón, en el tercer recuadro deberá aparecer la suma de los números introducidos en los dos primeros recuadros.
- Un programa que muestre una ventana con un recuadro de texto y 3 etiquetas. En el recuadro de texto se escribirá un número (en sistema decimal). Cada vez que en el recuadro de texto se pulse una tecla, se mostrará en las 3 etiquetas de texto el equivalente de ese número en binario, octal y hexadecimal. Pista: en la ventana de propiedades, se deberá escoger "Eventos", para ver las posibles acciones relacionadas con el TextBox.

### 10.3. Usando ventanas predefinidas

En una aplicación basada en ventanas, típicamente tendremos que mostrar algún mensaje de aviso, o pedir una confirmación al usuario. Para ello podríamos crear un programa basado en múltiples ventanas, pero eso queda más allá de lo que pretende este texto.

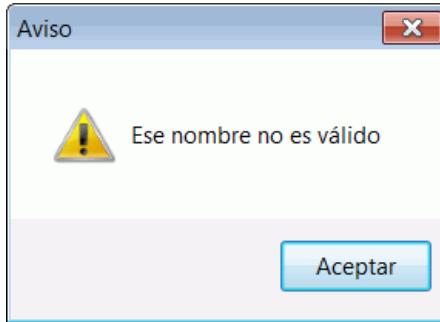
Una forma alternativa y sencilla de conseguirlo es usando "ventanas de mensaje". Éstas se pueden crear llamando a "MessageBox.Show", que tiene varias sintaxis posibles, según el número de parámetros que queramos utilizar. Por ejemplo, podemos mostrar un cierto texto de aviso en una ventana que tenga un título dado:

```
MessageBox.Show("Ese nombre no es válido", "Aviso");
```



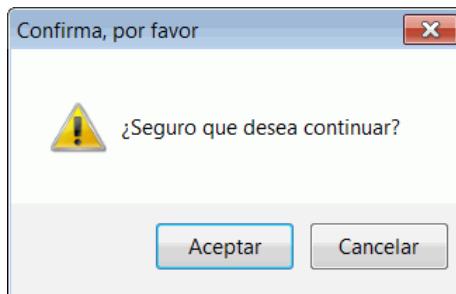
La segunda variante es indicar además qué botones queremos mostrar, y qué iconos de aviso:

```
MessageBox.Show("Ese nombre no es válido", "Aviso",
 MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
```



Y la tercera variante permite indicar además cual será el botón por defecto:

```
MessageBox.Show("¿Seguro que desea continuar?", "Confirma, por favor",
 MessageBoxButtons.OKCancel, MessageBoxIcon.Exclamation,
 MessageBoxDefaultButton.Button1);
```



Como se ve en estos ejemplos, tenemos algunos valores predefinidos para indicar qué botones o iconos queremos mostrar:

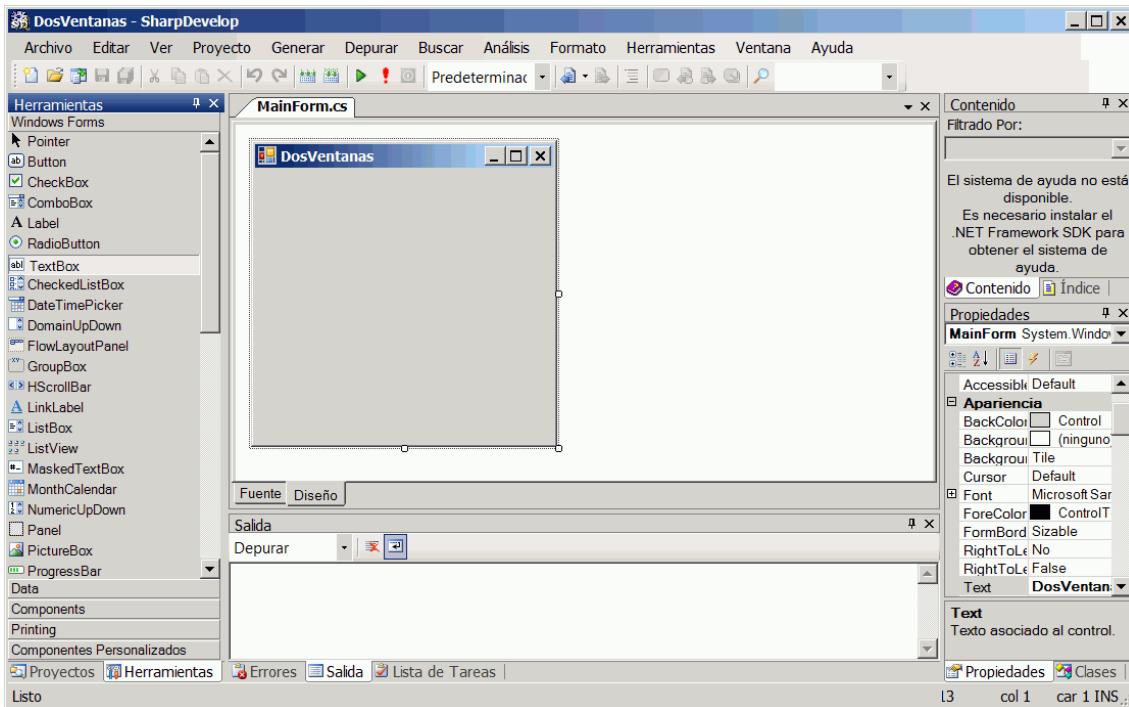
- Los botones (MessageBoxButtons) pueden ser: OK (Aceptar), OKCancel (Aceptar y Cancelar), AbortRetryIgnore (Anular, Reintentar y Omitir), YesNoCancel (Sí, No y Cancelar), YesNo (Sí y No), RetryCancel (Reintentar y Cancelar).
- Los iconos (MessageBoxIcon) pueden ser: None (ninguno), Hand (X blanca en un círculo con fondo rojo), Question (signo de interrogación en un círculo, no recomendado actualmente), Exclamation (signo de exclamación en un triángulo con fondo amarillo), Asterisk (letra 'i' minúscula en un círculo), Stop (X blanca en un círculo con fondo rojo), Error (X blanca en un círculo con fondo rojo), Warning (signo de exclamación en un triángulo con fondo amarillo), Information (letra 'i' minúscula en un círculo).
- Los botones (MessageBoxDefaultButton) por defecto pueden ser: Button1 (el primero), Button2 (el segundo), Button3 (el tercero).

Si queremos que el usuario responda tecleando, no tenemos ninguna ventana predefinida que nos lo permita (sí existe un "InputBox" en otros entornos de programación, como Visual Basic), así que deberíamos crear nosotros esa ventana de introducción de datos desde el editor visual o mediante código, elemento por elemento.

## 10.4. Una aplicación con dos ventanas

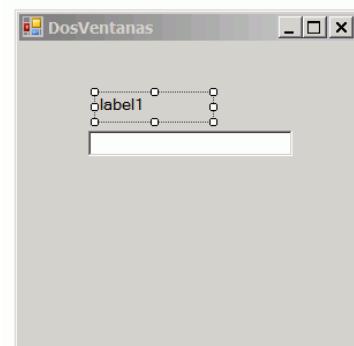
Si queremos una ventana auxiliar, que permita al usuario introducir más de un dato, deberemos crear un segundo formulario, y llamarlo desde la ventana principal. Vamos a ver los pasos con SharpDevelop (como siempre, serían muy similares en Visual Studio).

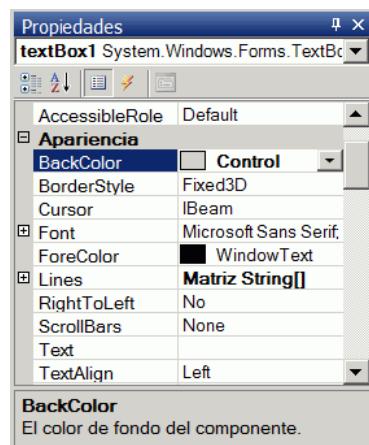
En primer lugar, crearemos un nuevo proyecto ("solución"), de tipo "Aplicación de Windows", y entraremos a la vista de diseño para crear la que será la ventana principal de nuestra aplicación:



Vamos a crear una ventana principal que tenga una casilla de texto (TextBox) en la que no se podrá escribir, sino que sólo se mostrarán resultados, y un botón que hará que aparezca la ventana secundaria, en la que sí podremos introducir datos.

Comenzamos por crear el "TextBox" que mostrará el texto, y el "Label" que aclarará qué es ese texto. Cambiamos el "Text" de la etiqueta para que muestre "Nombre y apellidos", y cambiamos la propiedad "ReadOnly" de la casilla de texto para que sea "true" (verdad), de modo que no se pueda escribir en esa casilla. También podemos cambiar el color de la casilla, para que sea más evidente que "no es una casilla normal". Por ejemplo, podemos hacer que sea gris, como el resto de la ventana. Para eso, cambiamos su propiedad BackColor (color de fondo). Es recomendable no usar colores prefijados, como el "gris", sino colores de la paleta de Windows, de modo que los elementos cambien correctamente si el usuario elige otra combinación de colores para el sistema operativo. Como el color de fondo de la ventana es "Control" (el color que tengan los controles de Windows), para la casilla de texto, escogeríamos también el color "Control", así:

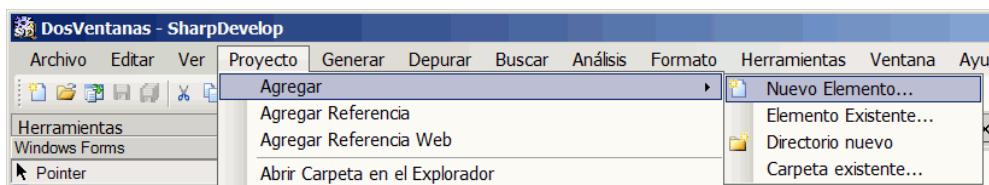




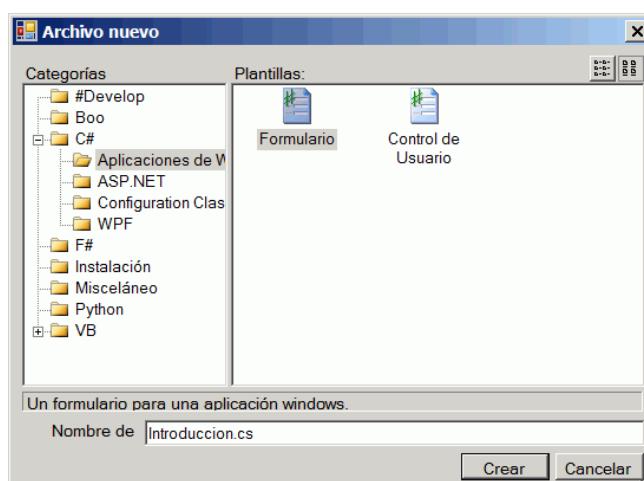
La apariencia de nuestra ventana debería ser parecida a ésta:



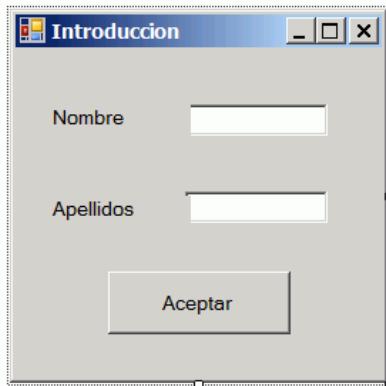
Para crear el segundo formulario (la ventana auxiliar), usamos la opción "Agregar / Nuevo Elemento", del menú "Proyecto":



y escogemos un nuevo Formulario, dentro de la pestaña de Aplicaciones de Windows:



En esta nueva ventana, entramos a la vista de diseño y añadimos dos casillas de texto (TextBox), con sus etiquetas aclaratorias (Label), y el botón de Aceptar:



Llega el momento de añadir el código a nuestro programa. Por una parte, haremos que el botón "Aceptar" cierre la ventana. Lo podemos conseguir con doble clic, para que nos aparezca la función que se lanzará con el suceso Click del botón (cuando se pulse el ratón sobre él), y añadimos la orden "Close()" en el cuerpo de esa función, así:

```
void Button1Click(object sender, EventArgs e)
{
 Close();
}
```

Además, para que desde la ventana principal se puedan leer los datos de ésta, podemos hacer que sus componentes sean públicos, o, mejor, crear un método "Get" que devuelva el contenido de estos componentes. Por ejemplo, podemos devolver el nombre y el apellido como parte de una única cadena de texto, así:

```
public string GetNombre()
{
 return textBox1.Text + " " + textBox2.Text;
}
```

Ya sólo falta que desde la ventana principal se muestre la ventana secundaria y se lean los valores al terminar. Para eso, añadimos un atributo en la ventana principal, que represente la ventana auxiliar:

```
public partial class MainForm : Form
{
 Introduccion ventanaIntro;
 ...
}
```

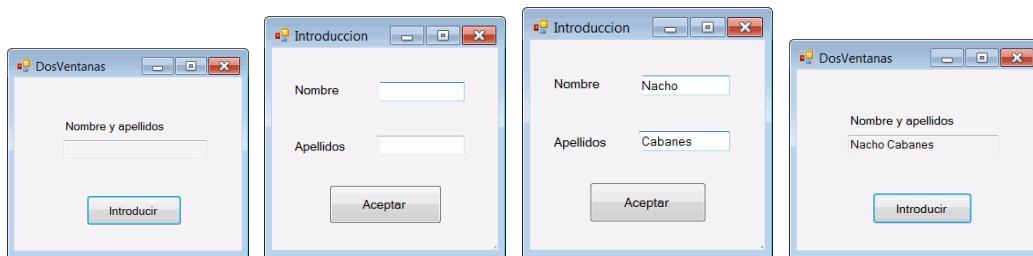
Y la inicializamos al final del constructor:

```
public MainForm()
{
 InitializeComponent();
 ventanaIntro = new Introduccion();
}
```

Finalmente, en el suceso Click del botón hacemos que se muestre la ventana secundaria usando `ShowDialog`, que espera a que se cierre ésta antes de permitirnos seguir trabajando en la ventana principal, y después leemos el valor que se había tecleado en dicha ventana:

```
void Button1Click(object sender, EventArgs e)
{
 ventanaIntro.ShowDialog();
 textBox1.Text = ventanaIntro.GetNombre();
}
```

El resultado sería una secuencia como esta:



## 10.5. Dibujando con Windows Forms

Windows es un entorno gráfico, por lo que se podría suponer que deberíamos tener la posibilidad de trabajar en "modo gráfico" desde dentro de Windows, dibujando líneas, círculos y demás figuras básicas. En efecto, podemos usar las posibilidades de "System.Drawing" para crear una ventana gráfica dentro de nuestro formulario (ventana de programa). Deberemos preparar también las "plumas" ("Pen", para los contornos) y las "brochas" ("Brush", para los rellenos) que queramos usar. Un ejemplo que dibujara una línea roja y una elipse azul cuando pulsemos un botón del formulario podría ser así:

```
void Button1Click(object sender, EventArgs e)
{
 // Creamos la pluma, el relleno y la ventana gráfica
 System.Drawing.Pen contornoRojo = new System.Drawing.Pen(
 System.Drawing.Color.Red);
 System.Drawing.SolidBrush rellenoAzul = new System.Drawing.SolidBrush(
 System.Drawing.Color.Blue);
 System.Drawing.Graphics ventanaGrafica;
 ventanaGrafica = this.CreateGraphics();

 // Dibujamos
 ventanaGrafica.DrawLine(contornoRojo, 200, 100, 300, 400);
 ventanaGrafica.FillEllipse(rellenoAzul, new Rectangle(0, 0, 200, 300));

 // Liberamos la memoria que habíamos reservado
 contornoRojo.Dispose();
 rellenoAzul.Dispose();
 ventanaGrafica.Dispose();
}
```

Los métodos para dibujar líneas, rectángulos, elipses, curvas, etc. son parte de la clase `Graphics`. Algunos de los métodos que ésta contiene y que pueden ser útiles para realizar dibujos sencillos son:

- `DrawArc`, para dibujar un arco.
- `DrawBezier`, para una curva spline de Bézier definida por cuatro puntos (estructuras `Point`).
- `DrawClosedCurve`, para una curva spline cerrada, a partir de un array de puntos.
- `DrawCurve`, para una curva.
- `DrawEllipse`, para dibujar una elipse, a partir del rectángulo que la contiene.
- `DrawIcon`, para dibujar una imagen representada por un ícono (`Icon`).
- `DrawImage`, para mostrar una imagen (`Image`).
- `DrawLine`, para una línea.
- `DrawPolygon`, para un polígono, a partir de un array de puntos.
- `DrawRectangle`, para un rectángulo.
- `DrawString`, para mostrar una cadena de texto.
- `FillEllipse`, para llenar el interior de una elipse.
- `FillPolygon`, para llenar el interior de un polígono.
- `FillRectangle`, para llenar el interior de un rectángulo.

Un ejemplo de cómo mostrar una imagen predefinida podría ser:

```
void Button2Click(object sender, EventArgs e)
{
 Graphics ventanaGrafica = this.CreateGraphics();
 Image imagen = new Bitmap("MiImagen.png");
 ventanaGrafica.DrawImage(imagen, 20, 20, 100, 90);
}
```

Esta imagen debería estar en la carpeta del programa ejecutable (que quizás no sea la misma que el fuente), y puede estar en formato BMP, GIF, PNG, JPG o TIFF.

Se puede encontrar más detalles en la referencia en línea (MSDN), por ejemplo en la página

[http://msdn.microsoft.com/es-es/library/system.drawing.graphics\\_methods.aspx](http://msdn.microsoft.com/es-es/library/system.drawing.graphics_methods.aspx)

## 10.6. Fecha y hora. Temporización

Desde C#, tenemos la posibilidad de manejar **fechas y horas** con facilidad. Para ello, tenemos el tipo de datos `DateTime`. Por ejemplo, podemos hallar la fecha (y hora) actual con:

```
DateTime fecha = DateTime.Now;
```

Dentro de ese tipo de datos `DateTime`, tenemos las herramientas para saber el día (Day), el mes (Month) o el año (Year) de una fecha, entre otros. También podemos calcular otras fechas sumando a la actual una cierta cantidad de segundos (AddSeconds), días (AddDays), etc. Un ejemplo básico de su uso sería:

```
/*
/* Ejemplo en C#
/* fechas.cs
/*
/*
/* Ejemplo básico de
/* manejo de fechas
/*
/*
/* Introducción a C#,
/* Nacho Cabanes
/*
*/
using System;

class ejemploFecha
{
 public static void Main()
 {
 DateTime fecha = DateTime.Now;
 Console.WriteLine("Hoy es {0} del mes {1} de {2}",
 fecha.Day, fecha.Month, fecha.Year);
 DateTime manyana = fecha.AddDays(1);
 Console.WriteLine("Mañana será {0}",
 manyana.Day);
 }
}
```

Algunas de las propiedades más útiles son: Now (fecha y hora actual de este equipo), Today (fecha actual); Day (día del mes), Month (número de mes), Year (año); Hour (hora), Minute (minutos), Second (segundos), Millisecond (milisegundos); DayOfWeek (día de la semana: su nombre en inglés); DayOfYear (día del año).

Y para calcular nuevas fechas, podemos usar métodos que generan un nuevo objeto DateTime, como: AddDays, AddHours, AddMilliseconds, AddMinutes, AddMonths, AddSeconds, AddHours, o bien un Add más genérico (para sumar una fecha a otra) y un Subtract también genérico (para restar una fecha de otra).

Cuando restamos dos fechas, obtenemos un dato de tipo "intervalo de tiempo" (TimeSpan), del que podemos saber detalles como la cantidad de días (sin decimales, Days, o con decimales, TotalDays), como se ve en este ejemplo:

```
/*
/* Ejemplo en C#
/* restarfechas.cs
/*
/*
/* Ejemplo ampliado de
/* manejo de fechas
/*
/*
/* Introducción a C#,
/* Nacho Cabanes
/*
*/
using System;
```

```

class ejemploFecha
{
 public static void Main()
 {
 DateTime fechaActual = DateTime.Now;
 DateTime fechaNacimiento = new DateTime(1990, 9, 18);

 TimeSpan diferencia =
 fechaActual.Subtract(fechaNacimiento) ;

 Console.WriteLine("Han pasado {0} días",
 diferencia.Days);

 Console.WriteLine("Si lo quieres con decimales: {0} días",
 diferencia.TotalDays);

 Console.WriteLine("Y si quieres más detalles: {0}",
 diferencia);
 }
}

```

El resultado de este programa sería algo como

```

Han pasado 7170 días
Si lo quieres con decimales: 7170,69165165654 días
Y si quieres más detalles: 7170.16:35:58.7031250

```

También podemos hacer una **pausa** en la ejecución: Si necesitamos que nuestro programa se detenga una cierta cantidad de tiempo, no hace falta que usemos un "while" que compruebe la hora continuamente, sino que podemos "bloquear" (Sleep) el subproceso (o hilo, "Thread") que representa nuestro programa una cierta cantidad de milésimas de segundo con: Thread.Sleep(5000);

Este método pertenece a System.Threading, que deberíamos incluir en nuestro apartado "using", o bien usar la llamada completa: System.Threading.Thread.Sleep(100);

## 10.7. Lectura de directorios

Si queremos analizar el contenido de un directorio, podemos emplear las clases Directory y DirectoryInfo.

La clase Directory contiene métodos para crear un directorio (.CreateDirectory), borrarlo (Delete), moverlo (Move), comprobar si existe (Exists), etc. Por ejemplo, podríamos hacer cosas como

```

string miDirectorio = @"c:\ejemplo1\ejemplo2";
if (!Directory.Exists(miDirectorio))
 Directory.CreateDirectory(miDirectorio);

```

También tenemos un método "GetFiles" que nos permite obtener la lista de ficheros que contiene un directorio. Así, podríamos listar todo el contenido de un directorio con:

```
string miDirectorio = @"c:\";
string[] listaFicheros;

listaFicheros = Directory.GetFiles(miDirectorio);
foreach(string fich in listaFicheros)
 Console.WriteLine(fich);
```

(la clase `Directory` está declarada en el espacio de nombres `System.IO`, por lo que deberemos añadirlo entre los "using" de nuestro programa).

La clase  `DirectoryInfo` permite obtener información sobre fechas de creación, modificación y acceso, y, de forma análoga,  `FileInfo` nos permite conseguir información similar sobre un fichero. Podríamos usar estas dos clases para ampliar el ejemplo anterior, y que no sólo muestre el nombre de cada fichero, sino otros detalles adicionales como el tamaño y la fecha de creación:

```
DirectoryInfo dir = new DirectoryInfo(miDirectorio);
FileInfo[] infoFicheros = dir.GetFiles();
foreach (FileInfo infoUnFich in infoFicheros)
{
 Console.WriteLine("{0}, de tamaño {1}, creado {2}",
 infoUnFich.Name,
 infoUnFich.Length,
 infoUnFich.CreationTime);
}
```

que escribiría cosas como

```
hiberfil.sys, de tamaño 1005113344, creado 15/12/2008 12:00:09
```

## **10.8. El entorno. Llamadas al sistema**

Si hay algo que no sepamos o podamos hacer, pero que alguna utilidad del sistema operativo sí es capaz de hacer por nosotros, podemos hacer que ella trabaje por nosotros. La forma de llamar a otras órdenes del sistema operativo (incluso programas externos de casi cualquier tipo) es creando un nuevo proceso con "`Process.Start`". Por ejemplo, podríamos lanzar el bloc de notas de Windows con:

```
Process proc = Process.Start("notepad.exe");
```

En los actuales sistemas operativos multitarea se da por sentado que no es necesario esperar a que termine otra la tarea, sino que nuestro programa puede proseguir. Si aun así, queremos esperar a que se complete la otra tarea, lo conseguiríamos con "`WaitForExit`", añadiendo esta segunda línea:

```
proc.WaitForExit();
```

## 10.9. Datos sobre "el entorno"

La clase "Environment" nos sirve para acceder a información sobre el sistema: unidades de disco disponibles, directorio actual, versión del sistema operativo y de la plataforma .Net, nombre de usuario y máquina, carácter o caracteres que se usan para avanzar de línea, etc:

```
string avanceLinea = Environment.NewLine;
Console.WriteLine("Directorio actual: {0}", Environment.CurrentDirectory);
Console.WriteLine("Nombre de la máquina: {0}", Environment.MachineName);
Console.WriteLine("Nombre de usuario: {0}", Environment.UserName);
Console.WriteLine("Dominio: {0}", Environment.UserDomainName);
Console.WriteLine("Código de salida del programa anterior: {0}",
 Environment.ExitCode);
Console.WriteLine("Línea de comandos: {0}", Environment.CommandLine);
Console.WriteLine("Versión del S.O.: {0}",
 System.Convert.ToString(Environment.OSVersion));
Console.WriteLine("Versión de .Net: {0}", Environment.Version.ToString());
String[] discos = Environment.GetLogicalDrives();
Console.WriteLine("Unidades lógicas: {0}", String.Join(", ", discos));
Console.WriteLine("Carpeta de sistema: {0}",
 Environment.GetFolderPath(Environment.SpecialFolder.System));
```

## 10.10. Acceso a bases de datos con SQLite

SQLite es un gestor de bases de datos de pequeño tamaño, que emplea SQL para las consultas, y del que existe una versión que se distribuye como un fichero DLL que acompañará al ejecutable de nuestro programa, o bien como un fichero en C que se podría integrar directamente en dicho ejecutable (si usamos lenguaje C en nuestro proyecto).

Para acceder a SQLite desde C#, tenemos disponible alguna adaptación de la biblioteca original. Una de ellas es System.Data.SQLite, disponible en <http://sqlite.phxsoftware.com/>

Con ella, los pasos a seguir para leer información desde una base de datos serían:

- Crear una conexión a la base de datos, mediante un objeto de la clase SQLiteConnection, en cuyo constructor indicaremos detalles como la ruta del fichero, la versión de SQLite, y si el fichero se debe crear o ya existe.
- Con un objeto de la clase SQLiteCommand detallaremos cual es la orden SQL a ejecutar, y la lanzaremos con ExecuteReader.
- Con "Read", leeremos cada dato (devuelve un bool que indica si se ha conseguido leer correctamente), y accederemos a los campos de cada dato como parte de un array: dato[0] será el primer campo, dato[1] será el segundo y así sucesivamente.
- Finalmente cerraremos la conexión con Close:

```
using System;
using System.Data.SQLite; //Utilizamos la DLL
```

```

public class pruebaSQLite
{
 public static void Main()
 {
 // Creamos la conexión a la BD
 // El Data Source contiene la ruta del archivo de la BD
 SQLiteConnection conexion =
 new SQLiteConnection
 ("Data Source=prueba.sqlite;Version=3;New=False;Compress=True;");
 conexion.Open();

 // Lanzamos la consulta y preparamos la estructura para leer datos
 string consulta = "select * from gente";
 SQLiteCommand cmd = new SQLiteCommand(consulta, conexion);
 SQLiteDataReader datos = cmd.ExecuteReader();

 // Leemos los datos de forma repetitiva
 while (datos.Read())
 {
 string nombre = Convert.ToString(datos[0]);
 int edad = Convert.ToInt32(datos[1]);
 // Y los mostramos
 System.Console.WriteLine("Nombre: {0}, edad: {1}",
 nombre, edad);
 }

 // Finalmente, cerramos la conexión
 conexion.Close();
 }
}

```

Deberemos compilar con la versión 2 (o superior) de la plataforma .Net. En Mono, esto supone emplear el compilador "gmcs" en vez de "mcs":

```
gmcs pruebaSQLite.cs /r:System.Data.SQLite.dll
```

Y para usarlo necesitaremos la versión 2 (o superior) de la plataforma .Net, o bien cargarlo a través de Mono:

```
mono pruebaSQLite.exe
```

La base de datos de prueba se podría haber creado previamente desde el propio entorno de SQLite, o con algún gestor auxiliar, como la extensión de Firefox llamada SQLite Manager o como la utilidad SQLite Database Browser.

Otra alternativa es utilizar un fuente similar, que creara una base de datos y que introdujera en ella algún dato, lo que se podría hacer así:

```

using System;
using System.Data.SQLite; //Utilizamos la DLL

public class pruebaSQLite2

```

```

{
 public static void Main()
 {
 Console.WriteLine("Creando la base de datos...");

 // Creamos la conexion a la BD.
 // El Data Source contiene la ruta del archivo de la BD
 SQLiteConnection conexion =
 new SQLiteConnection
 ("Data Source=prueba.sqlite;Version=3;New=True;Compress=True;");
 conexion.Open();

 // Creamos la tabla
 string creacion = "CREATE TABLE gente "
 +"(codigo INT PRIMARY KEY, nombre VARCHAR(40), edad INT);";
 SQLiteCommand cmd = new SQLiteCommand(creacion, conexion);
 cmd.ExecuteNonQuery();

 // E insertamos dos datos
 string insercion = "INSERT INTO gente VALUES (1,'Juan',20);";
 cmd = new SQLiteCommand(insercion, conexion);
 int cantidad = cmd.ExecuteNonQuery();
 if (cantidad < 1)
 Console.WriteLine("No se ha podido insertar");

 insercion = "INSERT INTO gente VALUES (2,'Pedro',19);";
 cmd = new SQLiteCommand(insercion, conexion);
 cantidad = cmd.ExecuteNonQuery();
 if (cantidad < 1)
 Console.WriteLine("No se ha podido insertar");

 // Finalmente, cerramos la conexion
 conexion.Close();

 Console.WriteLine("Creada.");
 }
}

```

Nota: el primer ejemplo daba por sentado que en la tabla "gente" existían dos campos (o más), de los cuales el primero era el "nombre" y el segundo era la "edad"; en este segundo ejemplo, estos son el segundo y el tercer dato, respectivamente, porque el primero es un "código", incluido a modo de ejemplo de cómo se puede crear una clave primaria.

### **10.11. Juegos con Tao.SDL**

SDL es una conocida biblioteca para la realización de juegos, que está disponible para diversos sistemas operativos y que permite tanto dibujar imágenes como comprobar el teclado, el ratón o el joystick, así como reproducir sonidos.

Tao.SDL es una adaptación de esta librería, que permite emplearla desde C#. Se puede descargar desde <http://www.mono-project.com/Tao>

SDL no es una librería especialmente sencilla, y tampoco lo acaba de ser Tao.SDL, así que los fuentes siguientes pueden resultar difíciles de entender, a pesar de realizar tareas muy básicas. Por eso, muchas veces es preferible "ocultar" los detalles de SDL creando nuestras propias clases "Hardware", "Imagen", etc., como verás al final de este apartado.

### 10.11.1. Mostrar una imagen estática

Vamos a ver un primer ejemplo, básico pero completo, que muestre cómo entrar a modo gráfico, cargar una imagen, dibujarla en pantalla, esperar cinco segundos y volver al sistema operativo. Tras el fuente comentaremos las principales funciones.

```
/*
 * Ejemplo en C#
 */
/* sd101.cs */
/*
 */
/* Primer acercamiento */
/* a SDL */
/*
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes */
/*
 */

using Tao.Sdl;
using System; // Para IntPtr (puntero: imágenes, etc)

public class Sd101
{
 private static void Main()
 {
 short anchoPantalla = 800;
 short altoPantalla = 600;
 int bitsColor = 24;
 int flags = Sd1.SDL_HWSURFACE | Sd1.SDL_DOUBLEBUF | Sd1.SDL_ANYFORMAT;
 IntPtr pantallaOculta;

 // Inicializamos SDL
 Sd1.SDL_Init(Sd1.SDL_INIT_EVERYTHING);
 pantallaOculta = Sd1.SDL_SetVideoMode(
 anchoPantalla,
 altoPantalla,
 bitsColor,
 flags);

 // Indicamos que se recorte lo que salga de la pantalla oculta
 Sd1.SDL_Rect rect2 =
 new Sd1.SDL_Rect(0, 0, (short) anchoPantalla, (short) altoPantalla);
 Sd1.SDL_SetClipRect(pantallaOculta, ref rect2);

 // Cargamos una imagen
 IntPtr imagen;
 imagen = Sd1.SDL_LoadBMP("personaje.bmp");
 if (imagen == IntPtr.Zero) {
 System.Console.WriteLine("Imagen inexistente!");
 }
 }
}
```

```

 Environment.Exit(4);
 }

 // Dibujamos la imagen
 short x = 400;
 short y = 300;
 short anchoImagen = 50;
 short altoImagen = 50;
 Sd1.SDL_Rect origen = new Sd1.SDL_Rect(0,0,anchoImagen,altoImagen);
 Sd1.SDL_Rect dest = new Sd1.SDL_Rect(x,y,anchoImagen,altoImagen);
 Sd1.SDL_BlitSurface(imagen, ref origen, pantallaOculta, ref dest);

 // Mostramos la pantalla oculta
 Sd1.SDL_Flip(pantallaOculta);

 // Y esperamos 5 segundos
 System.Threading.Thread.Sleep(5000);

 // Finalmente, cerramos SDL
 Sd1.SDL_Quit();
}

}

```

Algunas ideas básicas:

- `SDL_Init` es la rutina de inicialización, que entra a modo gráfico, con cierto ancho y alto de pantalla, cierta cantidad de colores y ciertas opciones adicionales.
- El tipo `SDL_Rect` define un "rectángulo" a partir de su origen (`x` e `y`), su ancho y su alto, y se usa en muchas operaciones.
- `SDL_SetClipRect` indica la zona de recorte (clipping) del tamaño de la pantalla, para que no tengamos que preocuparnos por si dibujamos una imagen parcialmente (o completamente) fuera de la pantalla visible.
- `SDL_LoadBMP` carga una imagen en formato BMP (si sólo usamos SDL "puro", no podremos emplear otros tipos que permitan menores tamaños, como el JPG, o transparencia, como el PNG). El tipo de dato que se obtiene es un "IntPtr" (del que no daemos más detalles), y la forma de comprobar si realmente se ha podido cargar la imagen es mirando si el valor obtenido es `IntPtr.Zero` (y en ese caso, no se habría podido cargar) u otro distinto (y entonces la imagen se habría leído sin problemas).
- `SDL_BlitSurface` vuelca un rectángulo (`SDL_Rect`) sobre otro, y lo usamos para ir dibujando todas las imágenes en una pantalla oculta, y finalmente volcar toda esa pantalla oculta a la pantalla visible, con lo que se evitan parpadeos (es una técnica que se conoce como "doble buffer").
- `SDL_Flip` vuelca esa pantalla oculta a la pantalla visible (el paso final de ese "doble buffer").
- Para la pausa no hemos usado ninguna función de SDL, sino `Thread.Sleep`, que ya habíamos comentado en el apartado sobre temporización.
- `SDL_Quit` libera los recursos (algo que generalmente haríamos desde un destructor).

Para compilar este ejemplo usando Mono, deberemos:

- Teclear (o copiar y pegar) el fuente.
- Copiar en la misma carpeta los ficheros DLL (Tao.Sdl.Dll y SDL.Dll) y las imágenes (en este caso, "personaje.bmp").
- Compilar con:

```
mcs sd101.cs /r:Tao.Sdl.dll
```

Y si empleamos Visual Studio o SharpDevelop, tendremos que:

- Crear un proyecto de "aplicación de consola".
- Reemplazar nuestro programa principal por éste.
- Copiar el fichero Tao.Sdl.Dll a la carpeta de fuentes, y añadirlo a las referencias del proyecto (normalmente, pulsando el botón derecho del ratón en la vista de clases del proyecto y escogiendo la opción "Aregar referencia").
- Copiar en la carpeta de ejecutables (típicamente bin/debug) los ficheros DLL (Tao.Sdl.Dll y SDL.Dll) y las imágenes (en este caso, "personaje.bmp").
- Compilar y probar.

Si no vamos a usar imágenes comprimidas (PNG o JPG), ni tipos de letra TTF, ni sonidos en formato MP3, ni funciones adicionales de dibujo (líneas, recuadros, círculos, etc). No necesitaremos ninguna DLL adicional.

### 10.11.2. Una imagen que se mueve con el teclado

Un segundo ejemplo algo más detallado podría permitirnos mover el personaje con las flechas del teclado, a una velocidad de 50 fotogramas por segundo, así:

```
/*
 *----- Ejemplo en C#
 *----- sd102.cs
 *----- Segundo acercamiento
 *----- a SDL
 *----- Introduccion a C#,
 *----- Nacho Cabanes
 *----- */

using Tao.Sdl;
using System; // Para IntPtr (puntero: imágenes, etc)

public class Sd102
{
 private static void Main()
 {
 short anchoPantalla = 800;
 short altoPantalla = 600;
 int bitsColor = 24;
 int flags = Sd1.SDL_HWSURFACE | Sd1.SDL_DOUBLEBUF | Sd1.SDL_ANYFORMAT
```

```

 | Sd1.SDL_FULLSCREEN;
IntPtr pantallaOculta;

// Inicializamos SDL
Sd1.SDL_Init(Sd1.SDL_INIT_EVERYTHING);
pantallaOculta = Sd1.SDL_SetVideoMode(
 anchoPantalla,
 altoPantalla,
 bitsColor,
 flags);

// Indicamos que se recorte lo que salga de la pantalla oculta
Sd1.SDL_Rect rect2 =
 new Sd1.SDL_Rect(0,0, (short) anchoPantalla, (short) altoPantalla);
Sd1.SDL_SetClipRect(pantallaOculta, ref rect2);

// Cargamos una imagen
IntPtr imagen;
imagen = Sd1.SDL_LoadBMP("personaje.bmp");
if (imagen == IntPtr.Zero) {
 System.Console.WriteLine("Imagen inexistente!");
 Environment.Exit(4);
}

// Parte repetitiva
bool terminado = false;
short x = 400;
short y = 300;
short anchoImagen = 50;
short altoImagen = 50;
Sd1.SDL_Event suceso;
int numkeys;
byte[] teclas;

do
{
 // Comprobamos sucesos
 Sd1.SDL_PollEvent(out suceso);
 teclas = Sd1.SDL_GetKeyState(out numkeys);

 // Miramos si se ha pulsado alguna flecha del cursor
 if (teclas[Sd1.SDLK_UP] == 1)
 y -= 2;
 if (teclas[Sd1.SDLK_DOWN] == 1)
 y += 2;
 if (teclas[Sd1.SDLK_LEFT] == 1)
 x -= 2;
 if (teclas[Sd1.SDLK_RIGHT] == 1)
 x += 2;
 if (teclas[Sd1.SDLK_ESCAPE] == 1)
 terminado = true;

 // Borramos pantalla
 Sd1.SDL_Rect origen = new Sd1.SDL_Rect(0,0,
 anchoPantalla,altoPantalla);
 Sd1.SDL_FillRect(pantallaOculta, ref origen, 0);
 // Dibujamos en sus nuevas coordenadas
}

```

```

origen = new Sdl.SDL_Rect(0,0,anchoImagen,altoImagen);
Sdl.SDL_Rect dest = new Sdl.SDL_Rect(x,y,
 anchoImagen,altoImagen);
Sdl.SDL_BlitSurface(imagen, ref origen,
 pantallaOculto, ref dest);

// Mostramos la pantalla oculta
Sdl.SDL_Flip(pantallaOculto);

// Y esperamos 20 ms
System.Threading.Thread.Sleep(20);

} while (!terminado);

// Finalmente, cerramos SDL
Sdl.SDL_Quit();

}

```

Las diferencias de este fuente con el anterior son:

- Al inicializar, añadimos una nueva opción, Sdl.SDL\_FULLSCREEN, para que el "juego" se ejecute a pantalla completa, en vez de hacerlo en ventana.
- Usamos un bucle "do...while" para repetir hasta que se pulse la tecla ESC.
- SDL\_Event es el tipo de datos que se usa para comprobar "sucesos", como pulsaciones de teclas, o de ratón, o el uso del joystick.
- Con SDL\_PollEvent forzamos a que se mire qué sucesos hay pendientes de analizar.
- SDL\_GetKeyState obtenemos un array que nos devuelve el estado actual de cada tecla. Luego podemos mirar cada una de esas teclas accediendo a ese array con el nombre de la tecla en inglés, así: *teclas[Sdl.SDLK\_RIGHT]*
- SDL\_FillRect rellena un rectángulo con un cierto color. Es una forma sencilla de borrar la pantalla o parte de ésta.

### 10.11.3. Escribir texto

Si queremos escribir texto usando tipos de letra TrueType (los habituales en Windows y Linux), los cambios no son grandes:

Debemos incluir el fichero DLL llamado **SDL\_ttf.dll** en la carpeta del ejecutable de nuestro programa, así como el fichero TTF correspondiente a cada tipo de letra que queramos usar.

Tenemos que declarar un nuevo dato que será nuestro tipo de letra, del tipo genérico "IntPtr":

```
IntPtr tipoDeLetra;
```

También tenemos que inicializar **SdlTtf** después de la inicialización básica de **SDL**:

```
SdlTtf.TTF_Init();
```

Luego preparamos el tipo de letra que queremos usar, indicando a partir de qué fichero TTF y en qué tamaño:

```
// Un tipo de letra, en tamaño 18
tipoDeLetra = SdlTtf.TTF_OpenFont("FreeSansBold.ttf", 18);
if (tipoDeLetra == IntPtr.Zero) {
 System.Console.WriteLine("Tipo de letra inexistente: FreeSansBold.ttf!");
 Environment.Exit(5);
}
```

A continuación, podemos crear una imagen a partir de una frase:

```
// Y creamos una imagen a partir de ese texto
Sdl.SDL_Color colorAzulIntenso = new Sdl.SDL_Color(50, 50, 255);
string texto = "Texto de ejemplo";
IntPtr textoComoImagen = SdlTtf.TTF_RenderText_Solid(
 tipoDeLetra, texto, colorAzulIntenso);

if (textoComoImagen == IntPtr.Zero) {
 System.Console.WriteLine("No se pudo renderizar el texto");
 Environment.Exit(6);
}
```

Y podríamos dibujar esa imagen en cualquier parte de la pantalla, como cualquier otra imagen:

```
// Escribimos el texto, como imagen
origen = new Sdl.SDL_Rect(0,0,anchoPantalla,altoPantalla);
dest = new Sdl.SDL_Rect(200,100,anchoPantalla,altoPantalla);
Sdl.SDL_BlitSurface(textoComoImagen, ref origen,
 pantallaOculta, ref dest);
```

Como esta forma de trabajar puede resultar engorrosa, dentro de poco lo mejoraremos, creando una clase "Fuente" que nos oculte todos estos detalles y nos permita escribir texto de forma sencilla.

#### 10.11.4. Imágenes PNG y JPG

Las imágenes BMP ocupan mucho espacio, y no permiten características avanzadas, como la transparencia (aunque se podría imitar). Si queremos usar imágenes en formatos más modernos, como JPG o PNG, sólo tenemos que incluir unos cuantos ficheros DLL más y hacer un pequeño cambio en el programa.

Los nuevos ficheros que necesitamos son: SDL\_image.dll (el principal), libpng13.dll (para imágenes PNG), zlib1.dll (auxiliar para el anterior) y jpeg.dll (si queremos usar imágenes JPG).

En el fuente, sólo cambiaría la orden de cargar cada imagen, que no utilizaría Sdl.SDL\_LoadBMP sino SdlImage.IMG\_Load:

```
imagen = SdlImage.IMG_Load("personaje.png");
```

### 10.11.5. Un fuente más modular: el "bucle de juego"

Un fuente con SDL puede resultar difícil de leer, y más aún si no está estructurado, sino que tiene toda la lógica dentro de "Main" y va creciendo arbitrariamente. Por eso, suele ser preferible crear nuestras propias funciones que la oculten un poco: funciones como "Inicializar", "CargarImagen", "ComprobarTeclas", "DibujarImagenes", etc.

Un "bucle de juego clásico" tendría una apariencia similar a esta:

- Comprobar eventos (pulsaciones de teclas, clics o movimiento de ratón, uso de joystick...)
- Mover los elementos del juego (personaje, enemigos, fondos móviles)
- Comprobar colisiones entre elementos del juego (que pueden suponer perder vidas o energía, ganar puntos, etc)
- Dibujar todos los elementos en pantalla en su estado actual
- Hacer una pausa al final de cada "fotograma", para que la velocidad del juego sea la misma en cualquier ordenador, y, de paso, para ayudar a la multitarea del sistema operativo.

Por tanto, crearemos esas funciones, junto con otras auxiliares para inicializar el sistema, dibujar una imagen en pantalla oculta, escribir un texto en pantalla oculta... El resultado podría ser éste:

```
/*
 *----- Ejemplo en C#
 *----- sd105.cs
 *----- Quinto acercamiento a SDL
 *----- Introducción a C#, Nacho Cabanes
 *----- */

using Tao.Sdl;
using System; // Para IntPtr (puntero: imágenes, etc)

public class Juego
{
 short anchoPantalla = 800;
 short altoPantalla = 600;
 bool terminado = false;
 short x = 400;
 short y = 300;
 short anchoImagen = 50;
 short altoImagen = 50;
 Sd1.SDL_Event suceso;
 int numkeys;
 byte[] teclas;
 IntPtr pantallaOculto;
 IntPtr tipoDeLetra;
 IntPtr imagen;

 void inicializar()
```

```

{
 int bitsColor = 24;
 int flags = Sdl.SDL_HWSURFACE | Sdl.SDL_DOUBLEBUF | Sdl.SDL_ANYFORMAT
 | Sdl.SDL_FULLSCREEN;

 // Inicializamos SDL
 Sdl.SDL_Init(Sdl.SDL_INIT_EVERYTHING);
 pantallaOculta = Sdl.SDL_SetVideoMode(
 anchoPantalla,
 altoPantalla,
 bitsColor,
 flags);
 // Y SdTTf, para escribir texto
 SdlTtf.TTF_Init();

 // Indicamos que se recorte lo que salga de la pantalla oculta
 Sdl.SDL_Rect rect2 =
 new Sdl.SDL_Rect(0,0, (short) anchoPantalla, (short) altoPantalla);
 Sdl.SDL_SetClipRect(pantallaOculta, ref rect2);

 // Cargamos una imagen
 imagen = SdlImage.IMG_Load("personaje.png");
 if (imagen == IntPtr.Zero) {
 System.Console.WriteLine("Imagen inexistente: personaje.png!");
 Environment.Exit(4);
 }

 // Y un tipo de letra, en tamaño 18
 tipoDeLetra = SdlTtf.TTF_OpenFont("FreeSansBold.ttf", 18);
 if (tipoDeLetra == IntPtr.Zero) {
 System.Console.WriteLine("Tipo de letra inexistente: FreeSansBold.ttf!");
 Environment.Exit(5);
 }
}

void comprobarTeclas()
{
 // Comprobamos sucesos
 Sdl.SDL_PollEvent(out suceso);
 teclas = Sdl.SDL_GetKeyState(out numkeys);

 // Miramos si se ha pulsado alguna flecha del cursor
 if (teclas[Sdl.SDLK_UP] == 1)
 y -= 2;
 if (teclas[Sdl.SDLK_DOWN] == 1)
 y += 2;
 if (teclas[Sdl.SDLK_LEFT] == 1)
 x -= 2;
 if (teclas[Sdl.SDLK_RIGHT] == 1)
 x += 2;
 if (teclas[Sdl.SDLK_ESCAPE] == 1)
 terminado = true;
}

void comprobarColisiones()
{
}

```

```

 // Todavia no comprobamos colisiones con enemigos
 // ni con obstaculos
 }

void moverElementos()
{
 // Todavia no hay ningun elemento que se mueva solo
}

void dibujarElementos()
{
 // Borramos pantalla
 Sd1.SDL_Rect origen = new Sd1.SDL_Rect(0,0,
 anchoPantalla,altoPantalla);
 Sd1.SDL_FillRect(pantallaOculto, ref origen, 0);

 // Dibujamos la imagen en sus nuevas coordenadas
 dibujarImagenOculto(imagen,x,y, anchoImagen, altoImagen);

 // Escribimos el texto, como imagen
 escribirTextoOculto("Texto de ejemplo",
 /* Coordenadas */ 200,100,
 /* Colores */ 50, 50, 255,
 tipoDeLetra
);

 // Mostramos la pantalla oculta
 Sd1.SDL_Flip(pantallaOculto);
}

void pausaFotograma()
{
 // Esperamos 20 ms
 System.Threading.Thread.Sleep(20);
}

void dibujarImagenOculto(IntPtr imagen, short x, short y,
 short ancho, short alto)
{
 Sd1.SDL_Rect origen = new Sd1.SDL_Rect(0, 0, ancho, alto);
 Sd1.SDL_Rect dest = new Sd1.SDL_Rect(x, y, ancho, alto);
 Sd1.SDL_BlitSurface(imagen, ref origen, pantallaOculto, ref dest);
}

void escribirTextoOculto(string texto,
 short x, short y, byte r, byte g, byte b, IntPtr fuente)
{
 // Creamos una imagen a partir de ese texto
 Sd1.SDL_Color color = new Sd1.SDL_Color(r, g, b);
 IntPtr textoComoImagen = Sd1Ttf.TTF_RenderText_Solid(
 fuente, texto, color);
 if (textoComoImagen == IntPtr.Zero){
 System.Console.WriteLine("No se pudo renderizar el texto");
 Environment.Exit(6);
}

```

```

 }
 dibujarImagenOculta(textoComoImagen, x,y, (short) (800-x), (short) (600-y));
 }

 bool partidaTerminada()
 {
 return terminado;
 }

private static void Main()
{
 Juego j = new Juego();
 j.inicializar();

 // Bucle de juego
 do {
 j.comprobarTeclas();
 j.moverElementos();
 j.comprobarColisiones();
 j.dibujarElementos();
 j.pausaFotograma();
 } while (! j.partidaTerminada());

 // Finalmente, cerramos SDL
 Sdl.SDL_Quit();
}

}

```

### 10.11.6. Varias clases auxiliares

Cuando el proyecto es grande, no basta con que esté desglosado en funciones. En esos casos, es muy interesante analizar qué objetos interaccionan y plantear el programa como una serie de clases que interactúan. Eso permite que el proyecto esté formado por varios fuentes de menor tamaño, cada uno con una responsabilidad clara, y a su vez esto facilita repartir el trabajo entre varios programadores.

Si aplicamos este planteamiento a nuestro esqueleto de juego, podríamos crear clases auxiliares como:

- Hardware, para ocultar el acceso a pantalla y teclado... y de paso, al ratón y el joystick. Podría tener una función "inicializar" que entrara a modo gráfico, una "TeclaPulsada" para ver si se ha pulsado una tecla concreta, una "Pausa", funciones para escribir texto y dibujar imágenes en pantalla oculta, una función "BorrarPantallaOculta" y una "VisualizarPantallaOculta".
- Imagen, para representar una imagen estática, y simplificar su carga mediante un constructor.
- ElemGrafico (elemento gráfico), que amplíe las posibilidades de una "Imagen", permitiendo imágenes animadas, formadas por varios fotogramas, y que permita comprobar colisiones entre una figura y otra.

- Fuente, para representar un tipo de letra.
- Sonido, para permitir reproducir efectos sonoros y melodías, ya sea una vez o de forma continua.

Y esto podría simplificar el cuerpo del juego para que acabara siendo algo así:

```
/*
/* Ejemplo en C#
/* sdl06.cs
/*
/*
/* Sexto acercamiento
/* a SDL
/*
/*
/* Introduccion a C#,
/* Nacho Cabanes
/*
using Tao.Sdl;

public class Juego
{
 short anchoPantalla = 800;
 short altoPantalla = 600;
 bool terminado = false;
 short x = 400;
 short y = 300;
 Fuente tipoDeLetra;
 ElemGrafico personaje;

 void inicializar()
 {
 bool pantallaCompleta = false;
 Hardware.Inicializar(800, 600, 24, pantallaCompleta);

 personaje = new ElemGrafico("personaje.png");
 tipoDeLetra = new Fuente("FreeSansBold.ttf", 18);
 }

 void comprobarTeclas()
 {
 if (Hardware.TeclaPulsada(Hardware.TECLA_ARR))
 y -= 2;
 if (Hardware.TeclaPulsada(Hardware.TECLA_ABA))
 y += 2;
 if (Hardware.TeclaPulsada(Hardware.TECLA_IQZ))
 x -= 2;
 if (Hardware.TeclaPulsada(Hardware.TECLA_DER))
 x += 2;
 if (Hardware.TeclaPulsada(Hardware.TECLA_ESC))
 terminado = true;
 }

 void comprobarColisiones()
 {
 // Todavia no comprobamos colisiones con enemigos
 }
}
```

```

 // ni con obstaculos
 }

void moverElementos()
{
 // Todavia no hay ningun elemento que se mueva solo
}

void dibujarElementos()
{
 // Borramos pantalla
 Hardware.BorrarPantallaOculta(0,0,0);

 // Dibujamos el personaje y el texto
 personaje.DibujarOculta(x, y);
 Hardware.EscribirTextoOculta("Texto de ejemplo",
 /* Coordenadas */ 200,100,
 /* Colores */ 50, 50, 255,
 tipoDeLetra
);

 // Finalmente, mostramos la pantalla oculta
 Hardware.VisualizarOculta();
}

void pausaFotograma()
{
 // Esperamos 20 ms
 Hardware.Pausa(20);
}

bool partidaTerminada()
{
 return terminado;
}

private static void Main()
{
 Juego j = new Juego();
 j.inicializar();

 // Bucle de juego
 do {
 j.comprobarTeclas();
 j.moverElementos();
 j.comprobarColisiones();
 j.dibujarElementos();
 j.pausaFotograma();
 } while (! j.partidaTerminada());

 // Finalmente, cerramos SDL
 Sdl.SDL_Quit();
}

```

}

(¿Y cómo podrían ser esas clases Hardware, Imagen, etc? Tienes detalles en un apéndice al final de este texto).

### **10.12. Algunos servicios de red.**

Muchos de los servicios que podemos obtener de Internet o de una red local son accesibles de forma sencilla, típicamente enmascarados como si leyéramos de un fichero o escribiéramos en él.

*(Nota: en este apartado se asumirá que el lector entiende algunos conceptos básicos de redes, como qué es una dirección IP, qué significa "localhost" o qué diferencias hay entre el protocolo TCP y el UDP).*

Como primer ejemplo, vamos a ver cómo podríamos recibir una página web (por ejemplo, la página principal de "www.nachocabanes.com"), línea a línea como si se tratara de un fichero de texto (StreamReader), y mostrar sólo las líneas que contengan un cierto texto (por ejemplo, la palabra "Pascal"):

```
// Ejemplo de descarga y análisis de una web
// Muestra las líneas que contienen "Pascal"

using System;
using System.IO; // Para Stream
using System.Net; // Para System.Net.WebClient

public class DescargarWeb
{
 public static void Main()
 {
 WebClient cliente = new WebClient();
 Stream conexion = cliente.OpenRead("http://www.nachocabanes.com");
 StreamReader lector = new StreamReader(conexion);
 string linea;
 int contador = 0;
 while ((linea=lector.ReadLine()) != null)
 {
 contador++;
 if (linea.IndexOf("Pascal") >= 0)
 Console.WriteLine("{0}: {1}", contador, linea);
 }
 conexion.Close();
 }
}
```

El resultado de este programa sería algo como:

28: <li><a href="pascal/index.php">Pascal/Delphi</a></li>

```
54: | Pascal / Delphi
204: Pascal/Delphi
```

Otra posibilidad que tampoco es complicada (aunque sí algo más que ésta última) es la de comunicar dos ordenadores, para enviar información desde uno y recibirla desde el otro. Se puede hacer de varias formas. Una de ellas es usando directamente el protocolo TCP: emplearemos un TcpClient para enviar y un TcpListener para recibir, y en ambos casos trataremos los datos como un tipo especial de fichero binario, un NetworkStream:

```
// Ejemplo de envío y recepción de frases a través de la red

using System;
using System.IO; // Para Stream
using System.Text; // Para Encoding
using System.Net; // Para Dns, IPAddress
using System.Net.Sockets; // Para NetworkStream

public class DescargarWeb
{
 static string direccionPrueba = "localhost";
 static int puertoPrueba = 2112;

 private static void enviar(string direccion, int puerto, string frase)
 {
 TcpClient cliente = new TcpClient(direccion, puerto);
 NetworkStream conexion = cliente.GetStream();
 byte[] secuenciaLetras = Encoding.ASCII.GetBytes(frase);

 conexion.Write(secuenciaLetras, 0, secuenciaLetras.Length);

 conexion.Close();
 cliente.Close();
 }

 private static string esperar(string direccion, int puerto)
 {
 // Tratamos de hallar la primera IP que corresponde
 // a una dirección como "localhost"
 IPAddress direccionIP = Dns.Resolve(direccion).AddressList[0];
 // Comienza la espera de información
 TcpListener listener = new TcpListener(direccionIP,puerto);
 listener.Start();
 TcpClient cliente = listener.AcceptTcpClient();
 NetworkStream conexion = cliente.GetStream();
 StreamReader lector = new StreamReader(conexion);

 string frase = lector.ReadToEnd();

 cliente.Close();
 listener.Stop();
 }

 return frase;
}
```

```

public static void Main()
{
 Console.WriteLine("Pulse 1 para recibir o 2 para enviar");
 string respuesta = Console.ReadLine();

 if (respuesta == "2") // Enviar
 {
 Console.Write("Enviando... ");
 enviar(direccionPrueba, puertoPrueba, "Prueba de texto");
 Console.WriteLine("Enviado");
 }
 else // Recibir
 {
 Console.WriteLine("Esperando... ");
 Console.WriteLine(esperar(direccionPrueba, puertoPrueba));
 Console.WriteLine("Recibido");
 }
}
}

```

Cuando lanzáramos el programa, se nos preguntaría si queremos enviar o recibir:

Pulse 1 para recibir o 2 para enviar

Lo razonable es lanzar primero el proceso que espera para recibir, pulsando 1:

```

1
Esperando...

```

Entonces lanzaríamos otra sesión del mismo programa en el mismo ordenador (porque estamos conectando a la dirección "localhost"), y en esta nueva sesión escogeríamos la opción de Enviar (2):

```

Pulse 1 para recibir o 2 para enviar
2
Enviando...

```

Instantáneamente, en la primera sesión recibiríamos el texto que hemos enviado desde la segunda, y se mostraría en pantalla:

```

Prueba de texto
Recibido

```

Y la segunda sesión confirmaría que el envío ha sido correcto:

Enviando... Enviado

Esta misma idea se podría usar como base para programas más elaborados, que comunicaran diferentes equipos (en este caso, la dirección no sería "localhost", sino la IP del otro equipo), como podría ser un chat o cualquier juego multijugador en el que hubiera que avisar a otros jugadores de los cambios realizados por cada uno de ellos.

Esto se puede conseguir a un nivel algo más alto, usando los llamados "Sockets" en vez de los TcpClient, o de un modo "no fiable", usando el protocolo UDP en vez de TCP, pero nosotros no veremos más detalles de ninguno de ambos métodos.

# 11. Depuración, prueba y documentación de programas

## 11.1. Conceptos básicos sobre depuración

La depuración es el análisis de un programa para descubrir fallos. El nombre en inglés es "debug", porque esos fallos de programación reciben el nombre de "bugs" (bichos).

Para eliminar esos fallos que hacen que un programa no se comporte como debería, se usan unas herramientas llamadas "depuradores". Estos nos permiten avanzar paso a paso para ver cómo avanza realmente nuestro programa, y también nos dejan ver los valores de las variables.

Veremos como ejemplo el caso de Visual Studio 2008 Express.

## 11.2. Depurando desde VS2008 Express

Vamos a tomar como ejemplo la secuencia de operaciones que se propuso como ejercicio al final del apartado 2.1:

a=5; b=a+2; b-=3; c=-3; c\*=2; ++c; a\*=b;

Esto se convertiría en un programa como

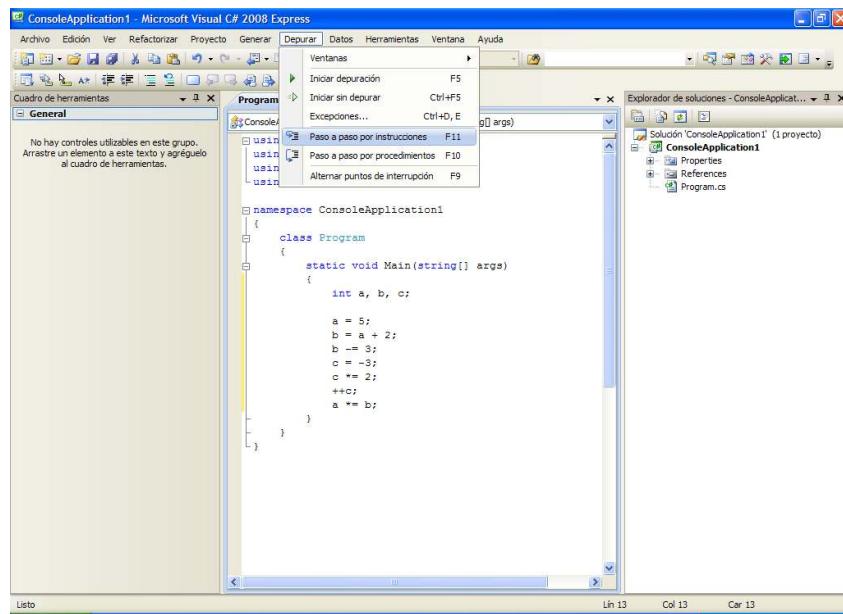
```
using System;

namespace ConsoleApplication1
{
 class Program
 {
 static void Main(string[] args)
 {
 int a, b, c;

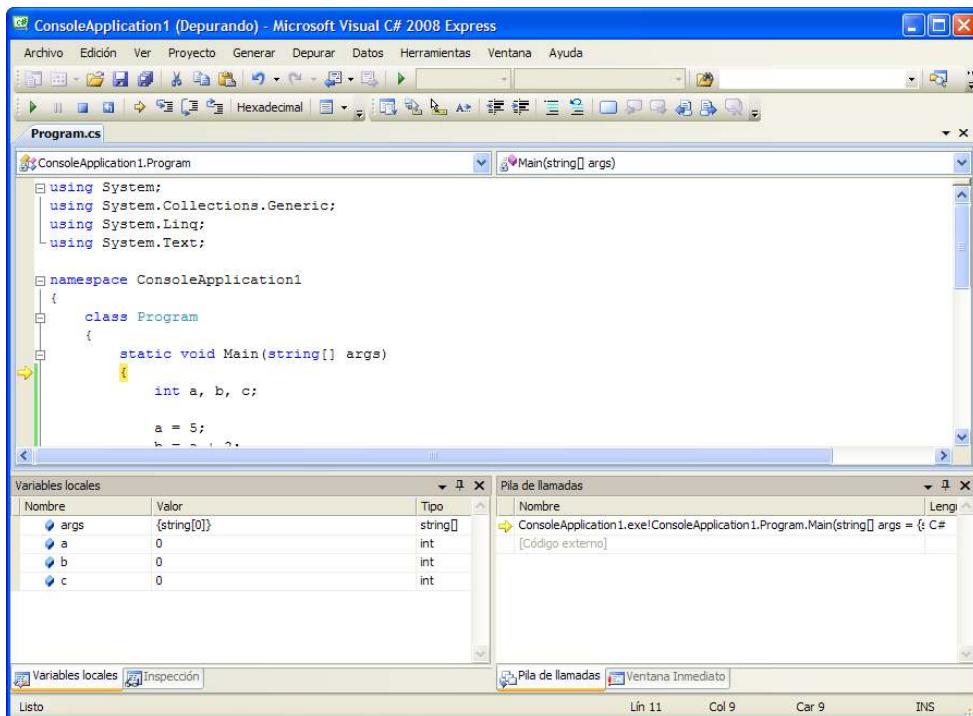
 a = 5;
 b = a + 2;
 b -= 3;
 c = -3;
 c *= 2;
 ++c;
 a *= b;
 }
 }
}
```

Ni siquiera necesitamos órdenes "WriteLine", porque no mostraremos nada en pantalla, será el propio entorno de desarrollo de Visual Studio el que nos muestre los valores de las variables.

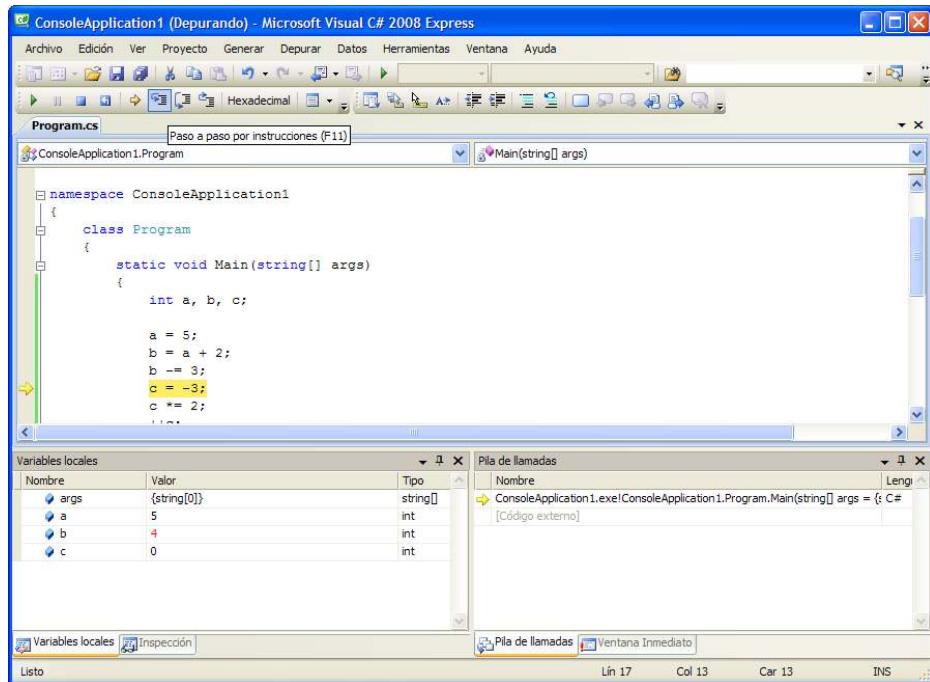
Para avanzar paso a paso y ver los valores de las variables, entramos al menú "Depurar". En él aparece la opción "Paso a paso por instrucciones" (al que corresponde la tecla F11):



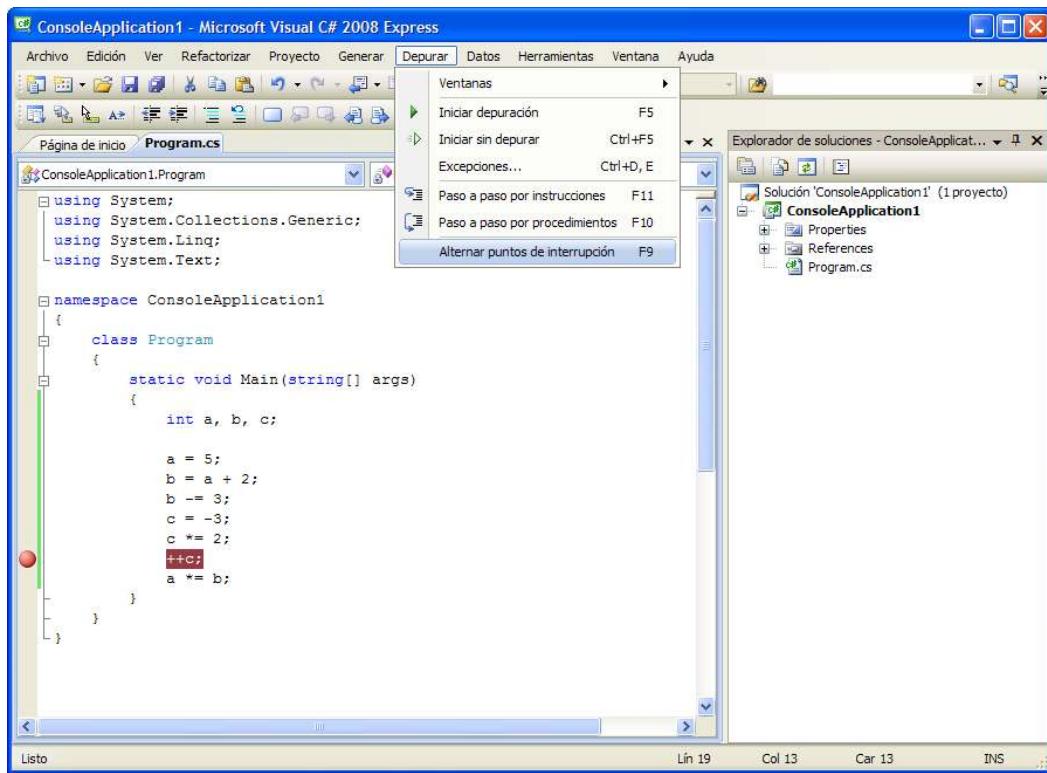
Si escogemos esa opción del menú o pulsamos F11, aparece una ventana inferior con la lista de variables, y un nuevo cursor amarillo, que se queda al principio del programa:



Cada vez que pulsemos nuevamente F11 (o vayamos al menú, o al botón correspondiente de la barra de herramientas), el depurador analiza una nueva línea de programa, muestra los valores de las variables correspondientes, y se vuelve a quedar parado, realizando con fondo amarillo la siguiente línea que se analizará:



Aquí hemos avanzado desde el principio del programa, pero eso no es algo totalmente habitual. Es más frecuente que supongamos en qué zona del programa se encuentra el error, y sólo queramos depurar una zona de programa. La forma de conseguirlo es escoger otra de las opciones del menú de depuración: "Alternar puntos de ruptura" (tecla F9). Aparecerá una marca granate en la línea actual:



Si ahora iniciamos la depuración del programa, saltará sin detenerse hasta ese punto, y será entonces cuando se interrumpa. A partir de ahí, podemos seguir depurando paso a paso como antes.

### 11.3. Prueba de programas

Es frecuente que la corrección de ciertos errores introduzca a su vez errores nuevos. Por eso, una forma habitual de garantizar la calidad de los programas es creando una "batería de pruebas" que permita comprobar de forma automática que todo se comporta como debería. Así, antes de dar por definitiva una versión de un programa, se lanza la batería de pruebas y se verifica que los resultados son los esperados.

Una forma sencilla de crear una batería de pruebas es comprobando los resultados de operaciones conocidas. Vamos a ver un ejemplo, para un programa que calcule las soluciones de una ecuación de segundo grado.

La fórmula que emplearemos es:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Un programa (incorrecto) para resolverlo podría ser:

```
public class SegundoGrado {

 public static void Resolver(
 float a, float b, float c,
 out float x1, out float x2)
 {

 float discriminante;
 discriminante = b*b - 4*a*c;

 if (discriminante < 0)
 {
 x1 = -9999;
 x2 = -9999;
 }
 else if (discriminante == 0)
 {
 x1 = - b / (2*a);
 x2 = -9999;
 }
 else
 {
 x1 = (float) ((- b + Math.Sqrt(discriminante)) / 2*a);
 x2 = (float) ((- b - Math.Sqrt(discriminante)) / 2*a);
 }
 }
}
```

```
}
```

Es decir, si alguna solución no existe, se devuelve un valor falso, que no sea fácil que se obtenga en un caso habitual, como -9999.

Y la batería de pruebas podría basarse en varios casos conocidos. Por ejemplo:

$$\begin{aligned}x^2 - 1 = 0 &\rightarrow x_1 = 1, x_2 = -1 \\x^2 = 0 &\rightarrow x_1 = 0, x_2 = \text{No existe (solución única)} \\x^2 - 3x = 0 &\rightarrow x_1 = 3, x_2 = 0 \\2x^2 - 2 = 0 &\rightarrow x_1 = 1, x_2 = -1\end{aligned}$$

Estos casos de prueba se podrían convertir en un programa como:

```
using System;

public class PruebaSegundoGrado {
 public static void Main(){
 float soluc1, soluc2;

 Console.WriteLine("Probando ecuaciones de segundo grado");

 SegundoGrado Ecuacion = new SegundoGrado();

 Console.Write("Probando x2 - 1 = 0 ...");
 SegundoGrado.Resolver((float)1, (float) 0, (float) -1,
 out soluc1, out soluc2);
 if ((soluc1 == 1) && (soluc2 == -1))
 Console.WriteLine("OK");
 else
 Console.WriteLine("Falla");

 Console.Write("Probando x2 = 0 ...");
 SegundoGrado.Resolver((float)1, (float)0, (float)0,
 out soluc1, out soluc2);
 if ((soluc1 == 0) && (soluc2 == -9999))
 Console.WriteLine("OK");
 else
 Console.WriteLine("Falla");

 Console.Write("Probando x2 - 3x = 0 ...");
 SegundoGrado.Resolver((float)1, (float)-3, (float)0,
 out soluc1, out soluc2);
 if ((soluc1 == 3) && (soluc2 == 0))
 Console.WriteLine("OK");
 else
 Console.WriteLine("Falla");

 Console.Write("Probando 2x2 - 2 = 0 ...");
 SegundoGrado.Resolver((float)2, (float)0, (float)-2,
 out soluc1, out soluc2);
```

```

 if ((soluc1 == 1) && (soluc2 == -1))
 Console.WriteLine("OK");
 else
 Console.WriteLine("Falla");
}
}

```

El resultado de este programa es:

```

Probando ecuaciones de segundo grado
Probando x2 - 1 = 0 ...OK
Probando x2 = 0 ...OK
Probando x2 -3x = 0 ...OK
Probando 2x2 - 2 = 0 ...Falla

```

Vemos que en uno de los casos, la solución no es correcta. Revisaríamos los pasos que da nuestro programa, corregiríamos el fallo y volveríamos a lanzar las pruebas automatizadas. En este caso, el problema es que falta un paréntesis, para dividir entre  $(2*a)$ , de modo que estamos diviendo entre 2 y luego multiplicando por a.

La ventaja de crear baterías de pruebas es que es una forma muy rápida de probar un programa, por lo que se puede aplicar tras cada pocos cambios para comprobar que todo es correcto. El inconveniente es que **NO GARANTIZA** que el programa sea correcto, sino sólo que no falla en ciertos casos. Por ejemplo, si la batería de pruebas anterior solo contuviera las tres primeras pruebas, no habría descubierto el fallo del programa.

Por tanto, las pruebas sólo permiten asegurar que el programa falla en caso de encontrar problemas, pero no permiten asegurar nada en caso de que no se encuentren problemas: puede que aun así exista un fallo que no hayamos detectado.

Para crear las baterías de pruebas, lo que más ayuda es la experiencia y el conocimiento del problema. Algunos expertos recomiendan que, si es posible, las pruebas las realice un equipo de desarrollo distinto al que ha realizado el proyecto principal, para evitar que se omitan cosas que se "den por supuestas".

#### **11.4. Documentación básica de programas**

La mayor parte de la documentación de un programa "serio" se debe realizar antes de comenzar a teclear:

- Especificaciones de requisitos, que reflejen lo que el programa debe hacer.
- Diagramas de análisis, como los diagramas de casos de uso UML, para plasmar de una forma gráfica lo que un usuario podrá hacer con el sistema.
- Diagramas de diseño, como los diagramas de clases UML que vimos en el apartado 6.4, para mostrar qué tipos de objetos formarán realmente nuestro programa y cómo interactuarán.
- ...

Casi todos esos diagramas caen fuera del alcance de este texto: en una introducción a la programación se realizan programas de pequeño tamaño, para los que no es necesaria una gran planificación.

Aun así, hay un tipo de documentación que sí debe estar presente en cualquier problema: los **comentarios** que aclaren todo aquello que no sea obvio.

Por eso, este apartado se va a centrar en algunas de las pautas que los expertos suelen recomendar para los comentarios en los programas, y también veremos como a partir de estos comentarios se puede generar documentación adicional de forma casi automática.

#### **11.4.1. Consejos para comentar el código**

Existen buenas recopilaciones de consejos en Internet. Yo voy a incluir (algo resumida) una de José M. Aguilar, recopilada en su blog "Variable not found". El artículo original se puede encontrar en:

<http://www.variablenotfound.com/2007/12/13-consejos-para-comentar-tu-código.html>

##### **1. Comenta a varios niveles**

Comenta los distintos bloques de los que se compone tu código, aplicando un criterio uniforme y distinto para cada nivel, por ejemplo:

- En cada clase, incluir una breve descripción, su autor y fecha de última modificación
- Por cada método, una descripción de su objeto y funcionalidades, así como de los parámetros y resultados obtenidos

(Lo importante es ceñirse a unas normas y aplicarlas siempre).

##### **2. Usa párrafos comentados**

Además, es recomendable dividir un bloque de código extenso en "párrafos" que realicen una tarea simple, e introducir un comentario al principio además de una línea en blanco para separar bloques:

```
// Comprobamos si todos los datos
// son correctos
foreach (Record record in records)
{
 ...
}
```

##### **3. Tabula por igual los comentarios de líneas consecutivas**

Si tenemos un bloque de líneas de código, cada una con un comentario, será más legible si están alineados:

```
const MAX_ITEMS = 10; // Número máximo de paquetes
const MASK = 0x1F; // Máscara de bits TCP
```

Ojo a las tabulaciones. Hay editores de texto que usan el carácter ASCII (9) y otros, lo sustituyen por un número determinado de espacios, que suelen variar según las preferencias

personales del desarrollador. Lo mejor es usar espacios simples o asegurarse de que esto es lo que hace el IDE correspondiente.

#### **4. No insultes la inteligencia del lector**

Debemos evitar comentarios absurdos como:

```
if (a == 5) // Si a vale cinco, ...
 contador = 0; // ... ponemos el contador a cero
...

```

#### **5. Sé correcto**

Evita comentarios del tipo "ahora compruebo que el estúpido usuario no haya introducido un número negativo", o "este parche corrige el efecto colateral producido por la patética implementación del inepto desarrollador inicial".

Relacionado e igualmente importante: cuida la ortografía.

#### **6. No pierdas el tiempo**

No comentes si no es necesario, no escribas nada más que lo que necesites para transmitir la idea. Nada de diseños realizados a base de caracteres ASCII, ni florituras, ni chistes, ni poesías, ni chascarrillos.

#### **7. Utiliza un estilo consistente**

Hay quien opina que los comentarios deberían ser escritos para que los entendieran no programadores. Otros, en cambio, piensan que debe servir de ayuda para desarrolladores exclusivamente. En cualquier caso, lo que importa es que siempre sea de la misma forma, orientados al mismo destinatario.

#### **8. Para los comentarios internos usa marcas especiales**

Y sobre todo cuando se trabaja en un equipo de programación. El ejemplo típico es el comentario TODO (to-do, por hacer), que describe funciones pendientes de implementar:

```
int calcula(int x, int y)
{
 // TODO: implementar los cálculos
 return 0;
}
```

#### **9. Comenta mientras programas**

Ve introduciendo los comentarios conforme vas codificando. No lo dejes para el final, puesto que entonces te costará más del doble de tiempo, si es que llegas a hacerlo. Olvida las posturas "no tengo tiempo de comentar, voy muy apurado", "el proyecto va muy retrasado"... son simplemente excusas.

Hay incluso quien opina que los comentarios que describen un bloque deberían escribirse antes de codificarlo, de forma que, en primer lugar, sirvan como referencia para saber qué es lo que hay que hacer y, segundo, que una vez codificado éstos queden como comentarios para la posteridad. Un ejemplo:

```

public void ProcesaPedido()
{
 // Comprobar que hay material
 // Comprobar que el cliente es válido
 // Enviar la orden a almacén
 // Generar factura
}

```

**10. Comenta como si fuera para tí mismo. De hecho, lo es.**

A la hora de comentar no pienses sólo en mantenimiento posterior, ni creas que es un regalo que dejas para la posteridad del que sólo obtendrá beneficios el desarrollador que en el futuro sea designado para corregir o mantener tu código. En palabras del genial Phil Haack, "tan pronto como una línea de código sale de la pantalla y volvemos a ella, estamos en modo mantenimiento de la misma"

**11. Actualiza los comentarios a la vez que el código**

De nada sirve comentar correctamente una porción de código si en cuanto éste es modificado no se actualizan también los comentarios. Ambos deben evolucionar paralelamente, o estaremos haciendo más difícil la vida del desarrollador que tenga que mantener el software, al darle pistas incorrectas.

**12. La regla de oro del código legible**

Es uno de los principios básicos para muchos desarrolladores: deja que tu código hable por sí mismo. Aunque se sospecha que este movimiento está liderado por programadores a los que no les gusta comentar su código ;-), es totalmente cierto que una codificación limpia puede hacer innecesaria la introducción de textos explicativos adicionales:

```

Console.WriteLine("Resultado: " +
new Calculator()
 .Set(0)
 .Add(10)
 .Multiply(2)
 .Subtract(4)
 .Get()
);

```

**13. Difunde estas prácticas entre tus colegas**

Aunque nosotros mismos nos beneficiamos inmediatamente de las bondades de nuestro código comentado, la generalización y racionalización de los comentarios y la creación código inteligible nos favorecerá a todos, sobre todo en contextos de trabajo en equipo.

**11.4.2. Generación de documentación a partir del código fuente.**

Conocemos los comentarios de bloque /\* hasta \*/ y los comentarios hasta final de línea (a partir de una doble barra // ).

Pero en algunos lenguajes modernos, como Java y C#, existe una posibilidad adicional que puede resultar muy útil: usar comentarios que nos ayuden a crear de forma automática cierta documentación del programa.

Esta documentación típicamente será una serie páginas HTML enlazadas, o bien varios ficheros XML.

Por ejemplo, la herramienta (gratuita) Doxygen genera páginas como ésta a partir de un fuente en C#:

**Referencia de la Clase Personaje**

Diagrama de herencias de Personaje

```

graph TD
 Personaje[Personaje] --> ElemGrafico[ElemGrafico]

```

Lista de todos los miembros.

**Métodos públicos**

|                               |
|-------------------------------|
| Personaje (Juego j)           |
| void <b>MoverDerecha</b> ()   |
| void <b>MoverIzquierda</b> () |
| void <b>MoverArriba</b> ()    |
| void <b>MoverAbajo</b> ()     |
| int <b>GetVidas</b> ()        |
| void <b>SetVidas</b> (int n)  |
| void <b>QuitarVida</b> ()     |
| void <b>Morir</b> ()          |

Animación cuando el personaje muere: ambulancia, etc.

**Descripción detallada**

**Personaje:** uno de los tipos de elementos graficos del juego

**Ver también:**  
ElemGrafico Juego

**Autor:**  
1-DAI 2008/09

**Documentación de las funciones miembro**

|                                   |
|-----------------------------------|
| void Personaje::Morir () [inline] |
|-----------------------------------|

Animación cuando el personaje muere: ambulancia, etc.

Generado para ElectroFreddy por **doxygen** 1.5.7.1

La forma de conseguirlo es empleando otros dos tipos de comentarios: comentarios de documentación en bloque (desde `/**` hasta `*/`) y los comentarios de documentación hasta final de línea (a partir de una triple barra `///`).

Lo habitual es que estos tipos de comentarios se utilicen al principio de cada clase y de cada método, así:

```
/**
 * Personaje: uno de los tipos de elementos graficos del juego
 *
 * @see ElemGrafico Juego
 * @author 1-DAI 2008/09
 */
```

```

public class Personaje : ElemGrafico
{
 // Mueve el personaje a la derecha, si es posible (sin obstáculos)
 public void MoverDerecha() {
 (...)

 (...)

 /// Animación cuando el personaje muere: ambulancia, etc.
 public void Morir(){
 (...)

 }
}

```

Además, es habitual que tengamos a nuestra disposición ciertas "palabras reservadas" para poder afinar la documentación, como `@returns`, que da información sobre el valor que devuelve una función, o como `@see`, para detallar qué otras clases de nuestro programa están relacionadas con la actual.

Así, comparando el fuente anterior y la documentación que se genera a partir de él, podemos ver que:

- El comentario de documentación inicial, creado antes del comienzo de la clase, aparece en el apartado "Descripción detallada".
- El comentario de documentación del método "Morir" se toma como descripción de dicha función miembro.
- La función "MoverDerecha" también tiene un comentario que la precede, pero está con el formato de un comentario "normal" (doble barra), por lo que no se tiene en cuenta al generar la documentación.
- "`@author`" se usa para el apartado "Autor" de la documentación.
- "`@see`" se emplea en el apartado "Ver también", que incluye enlaces a la documentación de otros ficheros relacionados.

En el lenguaje Java, documentación como esta se puede crear con la herramienta `JavaDoc`, que es parte de la distribución "oficial" del lenguaje. En cambio, en C#, la herramienta estándar genera documentación en formato XML, que puede resultar menos legible, pero se pueden emplear alternativas gratuitas como `Doxygen`.

# Apéndice 1. Unidades de medida y sistemas de numeración

## Ap1.1. bytes, kilobytes, megabytes...

En informática, la unidad básica de información es el **byte**. En la práctica, podemos pensar que un byte es el equivalente a una **letra**. Si un cierto texto está formado por 2000 letras, podemos esperar que ocupe unos 2000 bytes de espacio en nuestro disco.

Eso sí, suele ocurrir que realmente un texto de 2000 letras que se guarde en el ordenador ocupe más de 2000 bytes, porque se suele incluir información adicional sobre los tipos de letra que se han utilizado, cursivas, negritas, márgenes y formato de página, etc.

Un byte se queda corto a la hora de manejar textos o datos algo más largos, con lo que se recurre a un múltiplo suyo, el **kilobyte**, que se suele abbreviar **Kb** o **K**.

En teoría, el prefijo kilo querría decir "mil", luego un kilobyte debería ser 1000 bytes, pero en los ordenadores conviene buscar por comodidad una potencia de 2 (pronto veremos por qué), por lo que se usa  $2^{10} = 1024$ . Así, la equivalencia exacta es 1 K = 1024 bytes.

Los K eran unidades típicas para medir la memoria de ordenadores: 640 K ha sido mucho tiempo la memoria habitual en los IBM PC y similares. Por otra parte, una página mecanografiada suele ocupar entre 2 K (cerca de 2000 letras) y 4 K.

Cuando se manejan datos realmente extensos, se pasa a otro múltiplo, el **megabyte** o Mb, que es 1000 K (en realidad 1024 K) o algo más de un millón de bytes. Por ejemplo, en un diskette "normal" caben 1.44 Mb, y en un Compact Disc para ordenador (Cd-Rom) se pueden almacenar hasta 700 Mb. La memoria principal (RAM) de un ordenador actual suele andar por encima de los 512 Mb, y un disco duro actual puede tener una capacidad superior a los 80.000 Mb.

Para estas unidades de gran capacidad, su tamaño no se suele medir en megabytes, sino en el múltiplo siguiente: en **gigabytes**, con la correspondencia 1 Gb = 1024 Mb. Así, son cada vez más frecuentes los discos duros con una capacidad de 120, 200 o más gigabytes.

Y todavía hay unidades mayores, pero que aún se utilizan muy poco. Por ejemplo, un **terabyte** son 1024 gigabytes.

Todo esto se puede resumir así:

| Unidad            | Equivalencia | Valores posibles                   |
|-------------------|--------------|------------------------------------|
| Byte              | -            | 0 a 255 (para guardar 1 letra)     |
| Kilobyte (K o Kb) | 1024 bytes   | Aprox. media página mecanografiada |
| Megabyte (Mb)     | 1024 Kb      | -                                  |
| Gigabyte (Gb)     | 1024 Mb      | -                                  |
| Terabyte (Tb)     | 1024 Gb      | -                                  |

## Ejercicios propuestos:

- ¿Cuántas letras se podrían almacenar en una agenda electrónica que tenga 32 Kb de capacidad?
- Si suponemos que una canción típica en formato MP3 ocupa cerca de 3.500 Kb, ¿cuántas se podrían guardar en un reproductor MP3 que tenga 256 Mb de capacidad?
- ¿Cuántos diskettes de 1,44 Mb harían falta para hacer una copia de seguridad de un ordenador que tiene un disco duro de 6,4 Gb? ¿Y si usamos compact disc de 700 Mb, cuántos necesitaríamos?
- ¿A cuantos CD de 700 Mb equivale la capacidad de almacenamiento de un DVD de 4,7 Gb? ¿Y la de uno de 8,5 Gb?

## **Ap1.2. Unidades de medida empleadas en informática (2): los bits**

Dentro del ordenador, la información se debe almacenar realmente de alguna forma que a él le resulte "cómoda" de manejar. Como la memoria del ordenador se basa en componentes electrónicos, la unidad básica de información será que una posición de memoria esté usada o no (totalmente llena o totalmente vacía), lo que se representa como un 1 o un 0. Esta unidad recibe el nombre de **bit**.

Un bit es demasiado pequeño para un uso normal (recordemos: sólo puede tener dos valores: 0 ó 1), por lo que se usa un conjunto de ellos, 8 bits, que forman un **byte**. Las matemáticas elementales (combinatoria) nos dicen que si agrupamos los bits de 8 en 8, tenemos 256 posibilidades distintas (variaciones con repetición de 2 elementos tomados de 8 en 8:  $VR_{2,8}$ ):

```
00000000
00000001
00000010
00000011
00000100
...
11111110
11111111
```

Por tanto, si en vez de tomar los bits de 1 en 1 (que resulta cómodo para el ordenador, pero no para nosotros) los utilizamos en grupos de 8 (lo que se conoce como un byte), nos encontramos con 256 posibilidades distintas, que ya son más que suficientes para almacenar una letra, o un signo de puntuación, o una cifra numérica o algún otro símbolo.

Por ejemplo, se podría decir que cada vez que encontramos la secuencia 00000010 la interpretaremos como una letra A, y la combinación 00000011 como una letra B, y así sucesivamente.

También existe una correspondencia entre cada grupo de bits y un número del 0 al 255: si usamos el sistema binario de numeración (que aprenderemos dentro de muy poco), en vez del sistema decimal, tenemos que:

|                                   |
|-----------------------------------|
| 0000 0000 (binario) = 0 (decimal) |
| 0000 0001 (binario) = 1 (decimal) |
| 0000 0010 (binario) = 2 (decimal) |
| 0000 0011 (binario) = 3 (decimal) |

...  
1111 1110 (binario) = 254 (decimal)  
1111 1111 (binario) = 255 (decimal)

En la práctica, existe un código estándar, el **código ASCII** (American Standard Code for Information Interchange, código estándar americano para intercambio de información), que relaciona cada letra, número o símbolo con una cifra del 0 al 255 (realmente, con una secuencia de 8 bits): la "a" es el número 97, la "b" el 98, la "A" el 65, la "B", el 32, el "0" el 48, el "1" el 49, etc. Así se tiene una forma muy cómoda de almacenar la información en ordenadores, ya que cada letra ocupará exactamente un byte (8 bits: 8 posiciones elementales de memoria).

Aun así, hay un inconveniente con el código ASCII: sólo los primeros 127 números son estándar. Eso quiere decir que si escribimos un texto en un ordenador y lo llevamos a otro, las letras básicas (A a la Z, 0 al 9 y algunos símbolos) no cambiarán, pero las letras internacionales (como la Ñ o las vocales con acentos) puede que no aparezcan correctamente, porque se les asignan números que no son estándar para todos los ordenadores.

**Nota:** Eso de que realmente el ordenador trabaja con ceros y unos, por lo que le resulta más fácil manejar los números que son potencia de 2 que los números que no lo son, es lo que explica que el prefijo *kilo* no quiera decir "exactamente mil", sino que se usa la potencia de 2 más cercana:  $2^{10} = 1024$ . Por eso, la equivalencia exacta es **1 K = 1024 bytes**.

## Apéndice 2. El código ASCII

El nombre del código ASCII viene de "American Standard Code for Information Interchange", que se podría traducir como Código Estándar Americano para Intercambio de Información.

La idea de este código es que se pueda compartir información entre distintos ordenadores o sistemas informáticos. Para ello, se hace corresponder una letra o carácter a cada secuencia de varios bits.

El código ASCII estándar es de 7 bits, lo que hace que cada grupo de bits desde el 0000000 hasta el 1111111 (0 a 127 en decimal) corresponda siempre a la misma letra.

Por ejemplo, en cualquier ordenador que use código ASCII, la secuencia de bits 1000001 (65 en decimal) corresponderá a la letra "A" (a, en mayúsculas).

Los códigos estándar "visibles" van del 32 al 127. Los códigos del 0 al 31 son códigos de control: por ejemplo, el carácter 7 (BEL) hace sonar un pitido, el carácter 13 (CR) avanza de línea, el carácter 12 (FF) expulsa una página en la impresora (y borra la pantalla en algunos ordenadores), etc.

Estos códigos ASCII estándar son:

|     | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
| 000 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS  | HT  |  |
| 010 | LF  | VT  | FF  | CR  | SO  | SI  | DLE | DC1 | DC2 | DC3 |  |
| 020 | DC4 | NAK | SYN | ETB | CAN | EM  | SUB | ESC | FS  | GS  |  |
| 030 | RS  | US  | SP  | !   | "   | #   | \$  | %   | &   | '   |  |
| 040 | (   | )   | *   | +   | ,   | -   | .   | /   | 0   | 1   |  |
| 050 | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | :   | ;   |  |
| 060 | <   | =   | >   | ?   | @   | A   | B   | C   | D   | E   |  |
| 070 | F   | G   | H   | I   | J   | K   | L   | M   | N   | O   |  |
| 080 | P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y   |  |
| 090 | Z   | [   | \   | ]   | ^   | _   | `   | a   | b   | c   |  |
| 100 | d   | e   | f   | g   | h   | í   | j   | k   | l   | m   |  |
| 110 | n   | o   | p   | q   | r   | s   | t   | u   | v   | w   |  |
| 120 | x   | y   | z   | {   |     | }   | ~   | .   |     |     |  |

Hoy en día, casi todos los ordenadores incluyen un código ASCII extendido de 8 bits, que permite 256 símbolos (del 0 al 255), lo que permite que se puedan usar también vocales acentuadas, eñes, y otros símbolos para dibujar recuadros, por ejemplo, aunque estos símbolos que van del número 128 al 255 no son estándar, de modo que puede que otro ordenador no los interprete correctamente si el sistema operativo que utiliza es distinto.

Una alternativa más moderna es el estándar Unicode, que permite usar caracteres de más de 8 bits (típicamente 16 bits, que daría lugar a 65.536 símbolos distintos), lo que permite que la transferencia de información entre distintos sistemas no tenga problemas de distintas interpretaciones.

## Apéndice 3. Sistemas de numeración.

### *Ap3.1. Sistema binario*

Nosotros normalmente utilizamos el **sistema decimal** de numeración: todos los números se expresan a partir de potencias de 10, pero normalmente lo hacemos sin pensar.

Por ejemplo, el número 3.254 se podría desglosar como:

$$3.254 = 3 \cdot 1000 + 2 \cdot 100 + 5 \cdot 10 + 4 \cdot 1$$

o más detallado todavía:

$$254 = 3 \cdot 10^3 + 2 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$$

(aunque realmente nosotros lo hacemos automáticamente: no nos paramos a pensar este tipo de cosas cuando sumamos o multiplicamos dos números).

Para los ordenadores no es cómodo contar hasta 10. Como partimos de "casillas de memoria" que están completamente vacías (0) o completamente llenas (1), sólo les es realmente cómodo contar con 2 cifras: 0 y 1.

Por eso, dentro del ordenador cualquier número se deberá almacenar como ceros y unos, y entonces los números se deberán desglosar en potencias de 2 (el llamado "**sistema binario**"):

$$13 = 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$$

o más detallado todavía:

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

de modo que el número decimal 13 se escribirá en binario como 1101.

En general, **convertir** un número binario al sistema decimal es fácil: lo expresamos como suma de potencias de 2 y sumamos:

$$\begin{aligned} 0110\ 1101 \text{ (binario)} &= 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = \\ &= 0 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 109 \text{ (decimal)} \end{aligned}$$

Convertir un número de decimal a binario resulta algo menos intuitivo. Una forma sencilla es ir dividiendo entre las potencias de 2, y coger todos los cocientes de las divisiones:

$$109 / 128 = 0 \text{ (resto: 109)}$$

$$109 / 64 = 1 \text{ (resto: 45)}$$

$$45 / 32 = 1 \text{ (resto: 13)}$$

$$13 / 16 = 0 \text{ (resto: 13)}$$

$$13 / 8 = 1 \text{ (resto: 5)}$$

$$5 / 4 = 1 \text{ (resto: 1)}$$

$1 / 2 = 0$  (resto: 1)  
 $1 / 1 = 1$  (se terminó).

Si "juntamos" los cocientes que hemos obtenido, aparece el número binario que buscábamos:  
109 decimal = 0110 1101 binario

(Nota: es frecuente separar los números binarios en grupos de 4 cifras -medio byte- para mayor legibilidad, como yo he hecho en el ejemplo anterior; a un grupo de 4 bits se le llama **nibble**).

Otra forma sencilla de convertir de decimal a binario es dividir consecutivamente entre 2 y coger los restos que hemos obtenido, pero en orden inverso:

109 / 2 = 54, resto 1  
54 / 2 = 27, resto 0  
27 / 2 = 13, resto 1  
13 / 2 = 6, resto 1  
6 / 2 = 3, resto 0  
3 / 2 = 1, resto 1  
1 / 2 = 0, resto 1  
(y ya hemos terminado)

Si leemos esos restos de abajo a arriba, obtenemos el número binario: 1101101 (7 cifras, si queremos completarlo a 8 cifras rellenamos con ceros por la izquierda: 01101101).

¿Y se pueden hacer operaciones con números binarios? Sí, casi igual que en decimal:

$$\begin{array}{llll} 0 \cdot 0 = 0 & 0 \cdot 1 = 0 & 1 \cdot 0 = 0 & 1 \cdot 1 = 1 \\ 0+0 = 0 & 0+1 = 1 & 1+0 = 1 & 1+1 = 10 \text{ (en decimal: 2)} \end{array}$$

### Ejercicios propuestos:

1. Expresar en sistema binario los números decimales 17, 101, 83, 45.
2. Expresar en sistema decimal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101
3. Sumar los números 01100110+10110010, 11111111+00101101. Comprobar el resultado sumando los números decimales obtenidos en el ejercicio anterior.
4. Multiplicar los números binarios de 4 bits 0100·1011, 1001·0011. Comprobar el resultado convirtiéndolos a decimal.

## Ap3.2. Sistema octal

Hemos visto que el sistema de numeración más cercano a como se guarda la información dentro del ordenador es el sistema binario. Pero los números expresados en este sistema de numeración "ocupan mucho". Por ejemplo, el número 254 se expresa en binario como 11111110 (8 cifras en vez de 3).

Por eso, se han buscado otros sistemas de numeración que resulten más "compactos" que el sistema binario cuando haya que expresar cifras medianamente grandes, pero que a la vez mantengan con éste una correspondencia algo más sencilla que el sistema decimal. Los más usados son el sistema octal y, sobre todo, el hexadecimal.

El sistema octal de numeración trabaja en base 8. La forma de convertir de decimal a binario será, como siempre dividir entre las potencias de la base. Por ejemplo:

$$\begin{aligned} 254 \text{ (decimal)} &\rightarrow \\ 254 / 64 &= 3 \text{ (resto: 62)} \\ 62 / 8 &= 7 \text{ (resto: 6)} \\ 6 / 1 &= 6 \text{ (se terminó)} \end{aligned}$$

de modo que

$$254 = 3 \cdot 8^2 + 7 \cdot 8^1 + 6 \cdot 8^0$$

o bien

$$254 \text{ (decimal)} = 376 \text{ (octal)}$$

Hemos conseguido otra correspondencia que, si bien nos resulta a nosotros más incómoda que usar el sistema decimal, al menos es más compacta: el número 254 ocupa 3 cifras en decimal, y también 3 cifras en octal, frente a las 8 cifras que necesitaba en sistema binario.

Pero además existe una correspondencia muy sencilla entre el sistema octal y el sistema binario: si agrupamos los bits de 3 en 3, el paso de **binario a octal** es rapidísimo

$$254 \text{ (decimal)} = 011\ 111\ 110 \text{ (binario)}$$

$$\begin{aligned} 011 \text{ (binario)} &= 3 \text{ (decimal y octal)} \\ 111 \text{ (binario)} &= 7 \text{ (decimal y octal)} \\ 110 \text{ (binario)} &= 6 \text{ (decimal y octal)} \end{aligned}$$

de modo que

$$254 \text{ (decimal)} = 011\ 111\ 110 \text{ (binario)} = 376 \text{ (octal)}$$

El paso desde el **octal al binario** y al decimal también es sencillo. Por ejemplo, el número 423 (octal) sería 423 (octal) = 100 010 011 (binario)

o bien

$$423 \text{ (octal)} = 4 \cdot 64 + 2 \cdot 8 + 3 \cdot 1 = 275 \text{ (decimal)}$$

De cualquier modo, el sistema octal no es el que más se utiliza en la práctica, sino el hexadecimal...

### Ejercicios propuestos:

1. Expresar en sistema octal los números decimales 17, 101, 83, 45.
2. Expresar en sistema octal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101
3. Expresar en el sistema binario los números octales 171, 243, 105, 45.
4. Expresar en el sistema decimal los números octales 162, 76, 241, 102.

### Ap3.3. Sistema hexadecimal

El sistema octal tiene un inconveniente: se agrupan los bits de 3 en 3, por lo que convertir de binario a octal y viceversa es muy sencillo, pero un byte está formado por 8 bits, que no es múltiplo de 3.

Sería más cómodo poder agrupar de 4 en 4 bits, de modo que cada byte se representaría por 2 cifras. Este sistema de numeración trabajará en base 16 ( $2^4 = 16$ ), y es lo que se conoce como sistema hexadecimal.

Pero hay una dificultad: estamos acostumbrados al sistema decimal, con números del 0 al 9, de modo que no tenemos cifras de un solo dígito para los números 10, 11, 12, 13, 14 y 15, que utilizaremos en el sistema hexadecimal. Para representar estas cifras usaremos las letras de la A a la F, así:

```

0 (decimal) = 0 (hexadecimal)
1 (decimal) = 1 (hexadecimal)
2 (decimal) = 2 (hexadecimal)
3 (decimal) = 3 (hexadecimal)
4 (decimal) = 4 (hexadecimal)
5 (decimal) = 5 (hexadecimal)
6 (decimal) = 6 (hexadecimal)
7 (decimal) = 7 (hexadecimal)
8 (decimal) = 8 (hexadecimal)
9 (decimal) = 9 (hexadecimal)
10 (decimal) = A (hexadecimal)
11 (decimal) = B (hexadecimal)
12 (decimal) = C (hexadecimal)
13 (decimal) = D (hexadecimal)
14 (decimal) = E (hexadecimal)
15 (decimal) = F (hexadecimal)

```

Con estas consideraciones, expresar números en el sistema hexadecimal ya no es difícil:

254 (decimal) ->  
 $254 / 16 = 15$  (resto: 14)  
 $14 / 1 = 14$  (se terminó)

de modo que

$$254 = 15 \cdot 16^1 + 14 \cdot 16^0$$

o bien

$$254 (\text{decimal}) = FE (\text{hexadecimal})$$

Vamos a repetirlo para un convertir de **decimal a hexadecimal** número más grande:

54331 (decimal) ->  
 $54331 / 4096 = 13$  (resto: 1083)  
 $1083 / 256 = 4$  (resto: 59)  
 $59 / 16 = 3$  (resto: 11)  
 $11 / 1 = 11$  (se terminó)

de modo que

$$54331 = 13 \cdot 4096 + 4 \cdot 256 + 3 \cdot 16 + 11 \cdot 1$$

o bien

$$254 = 13 \cdot 16^3 + 4 \cdot 16^2 + 3 \cdot 16^1 + 11 \cdot 16^0$$

es decir

$$54331 \text{ (decimal)} = D43B \text{ (hexadecimal)}$$

Ahora vamos a dar el paso inverso: convertir de **hexadecimal a decimal**, por ejemplo el número A2B5

$$A2B5 \text{ (hexadecimal)} = 10 \cdot 16^3 + 2 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0 = 41653$$

El paso de **hexadecimal a binario** también es (relativamente) rápido, porque cada dígito hexadecimal equivale a una secuencia de 4 bits:

|                 |   |              |   |                |
|-----------------|---|--------------|---|----------------|
| 0 (hexadecimal) | = | 0 (decimal)  | = | 0000 (binario) |
| 1 (hexadecimal) | = | 1 (decimal)  | = | 0001 (binario) |
| 2 (hexadecimal) | = | 2 (decimal)  | = | 0010 (binario) |
| 3 (hexadecimal) | = | 3 (decimal)  | = | 0011 (binario) |
| 4 (hexadecimal) | = | 4 (decimal)  | = | 0100 (binario) |
| 5 (hexadecimal) | = | 5 (decimal)  | = | 0101 (binario) |
| 6 (hexadecimal) | = | 6 (decimal)  | = | 0110 (binario) |
| 7 (hexadecimal) | = | 7 (decimal)  | = | 0111 (binario) |
| 8 (hexadecimal) | = | 8 (decimal)  | = | 1000 (binario) |
| 9 (hexadecimal) | = | 9 (decimal)  | = | 1001 (binario) |
| A (hexadecimal) | = | 10 (decimal) | = | 1010 (binario) |
| B (hexadecimal) | = | 11 (decimal) | = | 1011 (binario) |
| C (hexadecimal) | = | 12 (decimal) | = | 1100 (binario) |
| D (hexadecimal) | = | 13 (decimal) | = | 1101 (binario) |
| E (hexadecimal) | = | 14 (decimal) | = | 1110 (binario) |
| F (hexadecimal) | = | 15 (decimal) | = | 1111 (binario) |

de modo que A2B5 (hexadecimal) = 1010 0010 1011 0101 (binario)

y de igual modo, de **binario a hexadecimal** es dividir en grupos de 4 bits y hallar el valor de cada uno de ellos:

$$\begin{aligned} 110010100100100101010100111 &=> \\ 0110\ 0101\ 0010\ 0100\ 1010\ 1010\ 0111 &= 6524AA7 \end{aligned}$$

### Ejercicios propuestos:

1. Expresar en sistema hexadecimal los números decimales 18, 131, 83, 245.
2. Expresar en sistema hexadecimal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101
3. Expresar en el sistema binario los números hexadecimales 2F, 37, A0, 1A2.
4. Expresar en el sistema decimal los números hexadecimales 1B2, 76, E1, 2A.

### Ap3.4. Representación interna de los enteros negativos

Para los **números enteros negativos**, existen varias formas posibles de representarlos. Las más habituales son:

- **Signo y magnitud:** el primer bit (el de más a la izquierda) se pone a 1 si el número es negativo y se deja a 0 si es positivo. Los demás bits se calculan como ya hemos visto.

Por ejemplo, si usamos 4 bits, tendríamos

$$3 \text{ (decimal)} = 0011 \quad -3 = 1011$$

$$6 \text{ (decimal)} = 0110 \quad -6 = 1110$$

Es un método muy sencillo, pero que tiene el inconveniente de que las operaciones en las que aparecen números negativos no se comportan correctamente. Vamos a ver un ejemplo, con números de 8 bits:

$$13 \text{ (decimal)} = 0000\ 1101 \quad -13 \text{ (decimal)} = 1000\ 1101$$

$$34 \text{ (decimal)} = 0010\ 0010 \quad -34 \text{ (decimal)} = 1010\ 0010$$

$$13 + 34 = 0000\ 1101 + 0010\ 0010 = 0010\ 1111 = 47 \text{ (correcto)}$$

$$(-13) + (-34) = 1000\ 1101 + 1010\ 0010 = 0010\ 1111 = 47 \text{ (INCORRECTO)}$$

$$13 + (-34) = 0000\ 1101 + 1010\ 0010 = 1010\ 1111 = -47 \text{ (INCORRECTO)}$$

- **Complemento a 1:** se cambian los ceros por unos para expresar los números negativos.

Por ejemplo, con 4 bits

$$3 \text{ (decimal)} = 0011 \quad -3 = 1100$$

$$6 \text{ (decimal)} = 0110 \quad -6 = 1001$$

También es un método sencillo, en el que las operaciones con números negativos salen bien, y que sólo tiene como inconveniente que hay dos formas de expresar el número 0 (0000 0000 o 1111 1111), lo que complica algunos trabajos internos del ordenador.

**Ejercicio propuesto:** convertir los números decimales 13, 34, -13, -34 a sistema binario, usando complemento a uno para expresar los números negativos. Calcular (en binario) el resultado de las operaciones 13+34, (-13)+(-34), 13+(-34) y comprobar que los resultados que se obtienen son los correctos.

- **Complemento a 2:** para los negativos, se cambian los ceros por unos y se suma uno al resultado.

Por ejemplo, con 4 bits

$$3 \text{ (decimal)} = 0011 \quad -3 = 1101$$

$$6 \text{ (decimal)} = 0110 \quad -6 = 1010$$

Es un método que parece algo más complicado, pero que no es difícil de seguir, con el que las operaciones con números negativos salen bien, y no tiene problemas para expresar el número 0 (00000000).

**Ejercicio propuesto:** convertir los números decimales 13, 34, -13, -34 a sistema binario, usando complemento a dos para expresar los números negativos. Calcular (en

binario) el resultado de las operaciones  $13+34$ ,  $(-13)+(-34)$ ,  $13+(-34)$  y comprobar que los resultados que se obtienen son los correctos.

En general, todos los formatos que permiten guardar números negativos usan el primer bit para el signo. Por eso, si declaramos una variable como "unsigned", ese primer bit se puede utilizar como parte de los datos, y podemos almacenar números más grandes. Por ejemplo, un "unsigned int" en MsDos podría tomar valores entre 0 y 65.535, mientras que un "int" (con signo) podría tomar valores entre +32.767 y -32.768.

## Apéndice 4. Instalación de Visual Studio.

Visual Studio es la herramienta "oficial" desarrollada por Microsoft para permitirnos crear programas en lenguajes Visual Basic, C++ y, recientemente, C# (entre otros). Existe una versión gratuita, llamada Visual Studio Express, que puede ser una herramienta muy útil para el que está aprendiendo a programar y no puede permitirse adquirir la versión profesional. Por eso veremos algunos detalles sobre su instalación y su uso básico.

### Ap4.1. Visual Studio 2008 Express

Visual Studio 2008 no es ya la última versión de Visual Studio, pero tiene dos ventajas que aun así lo pueden hacer resultar atractivo:

- Necesita un ordenador menos potente (sobre todo en cuanto a cantidad de memoria) que la versión 2010.
- Existe versión en español, algo que en el momento de escribir este texto no ocurre para la versión 2010.

Nosotros usaremos Visual Studio 2008 en su versión Express, que es la que se puede usar libremente ("gratis"), y, por tanto, es la más razonablemente para principiantes, además de que necesita un ordenador menos potente que la versión profesional.

Se puede descargar de:

[http://www.microsoft.com/express/downloads/#Visual\\_Studio\\_2008\\_Express\\_Downloads](http://www.microsoft.com/express/downloads/#Visual_Studio_2008_Express_Downloads)

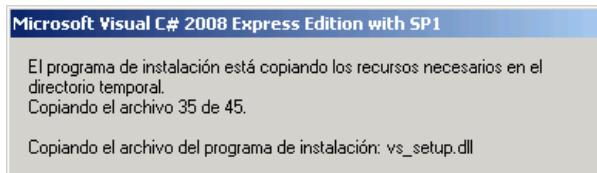
En concreto, en vez de descargar el paquete que llaman "Visual C#", y que es apenas un "descargador" para instalar con conexión a Internet, puede ser preferible descargar directamente la imagen ISO en español -Spanish- del DVD -900 Mb-, para poder instalar en cualquier ordenador y en cualquier momento, sin necesidad de estar conectado a Internet:

<http://www.microsoft.com/express/downloads/#2008-All>

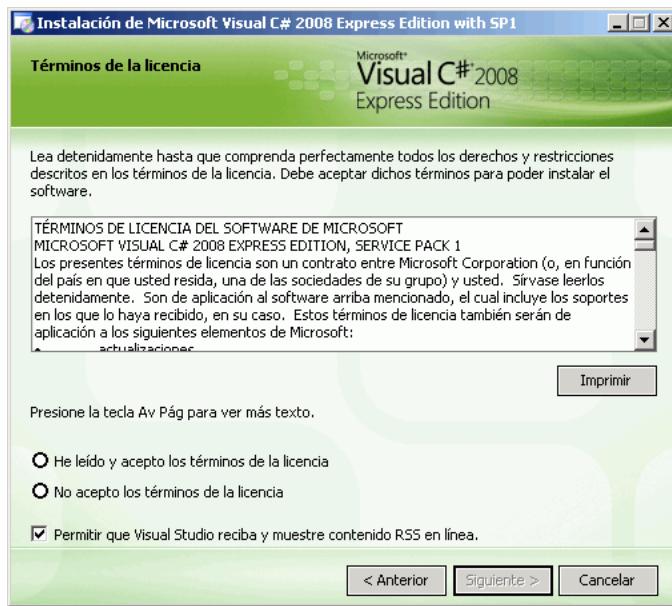
Esa imagen ISO se deberá grabar en DVD usando cualquier programa de grabación de CD y DVD. Al introducir y ejecutar el DVD deberá aparecer un menú como éste:



Escogemos Visual C#, y comenzarán a copiarse los ficheros temporales:



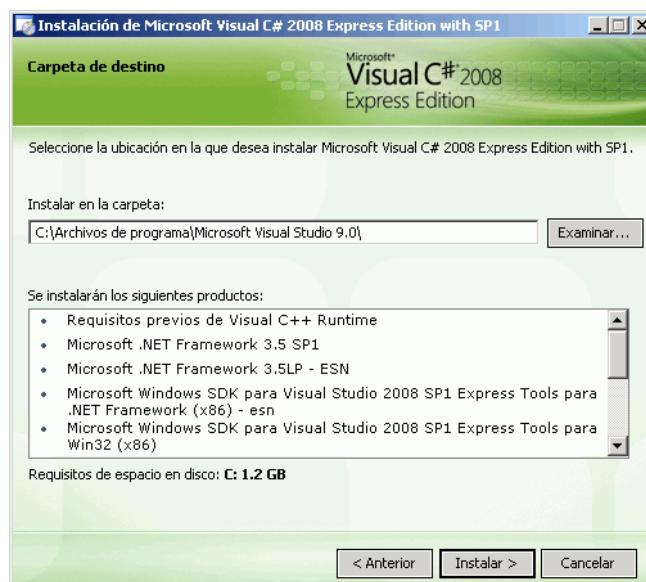
Después, como es habitual, deberemos aceptar el contrato de licencia:



E indicar si queremos algún componente adicional, como Silverlight (una alternativa al Flash de Adobe) o el gestor de bases de datos SQL Server, en su versión Express (para un uso de principiante, yo recomendaría no instalar ninguno de los dos):



Se nos preguntará en qué carpeta instalar



Y comenzará la instalación en sí:



Al final de la instalación se nos pedirá (posiblemente) reiniciar el equipo:



Y ya tendremos Visual C# disponible en nuestro menú de Inicio de Windows:



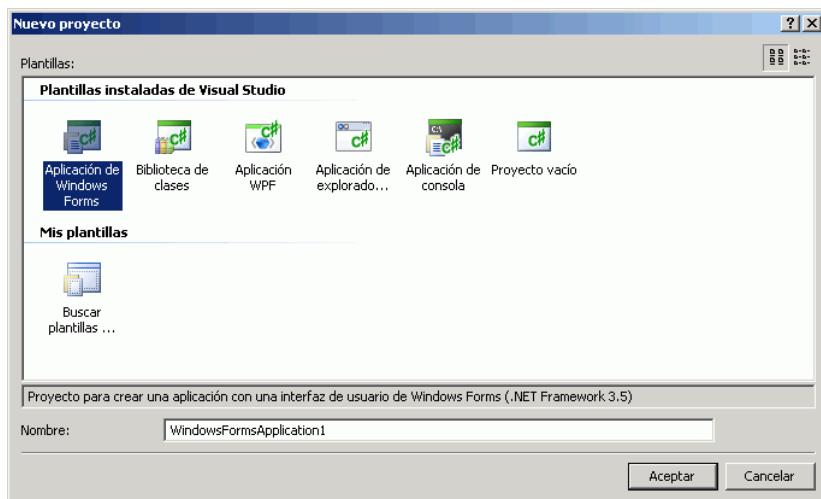
Cuando lo arrancamos, posiblemente nos aparecerá la página de inicio, con novedades propuestas por Microsoft:



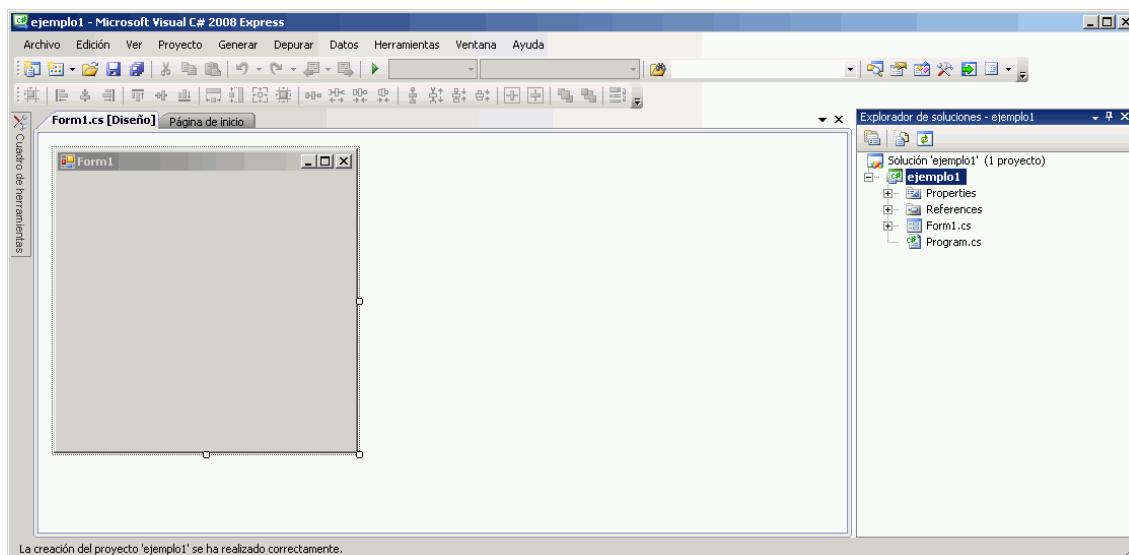
Y para empezar un proyecto nuevo, deberíamos ir al menú Archivo y escoger la opción "Nuevo Proyecto":



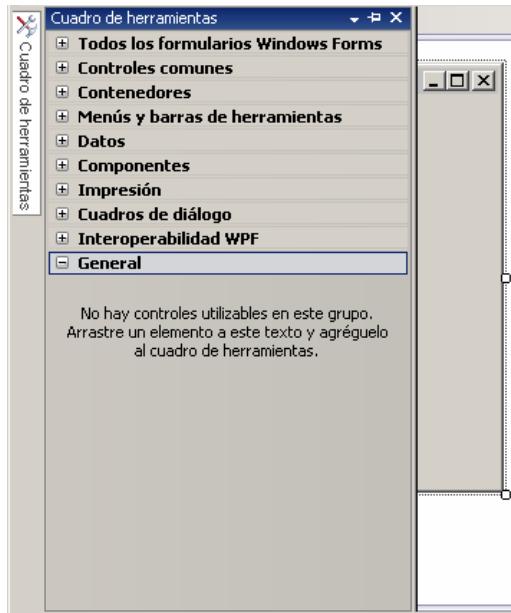
La mayoría de nuestros proyectos de principiante serían aplicaciones de consola. Los programas con "ventanitas" como la que hicimos en el apartado 8.2 serían "aplicaciones de Windows Forms":



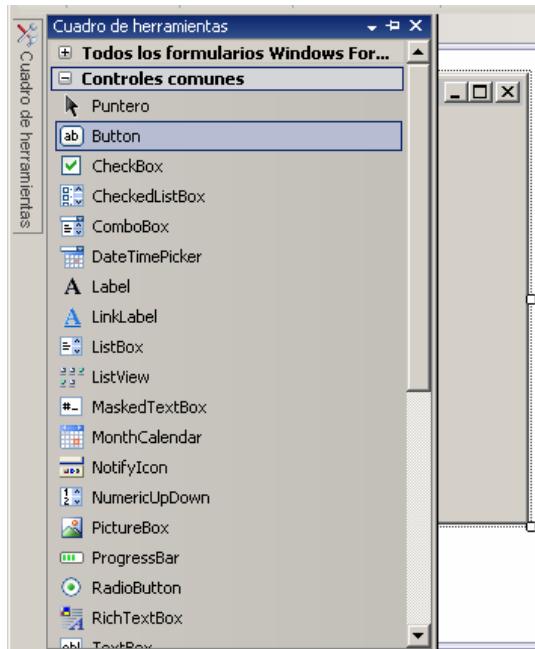
En la parte derecha de la pantalla aparecerán las clases que forman nuestro proyecto (por ahora, sólo "Program.cs", que es el esqueleto básico, "Form1.cs", que representa nuestra primera ventana), y en la parte izquierda aparecerá nuestra ventana vacía.



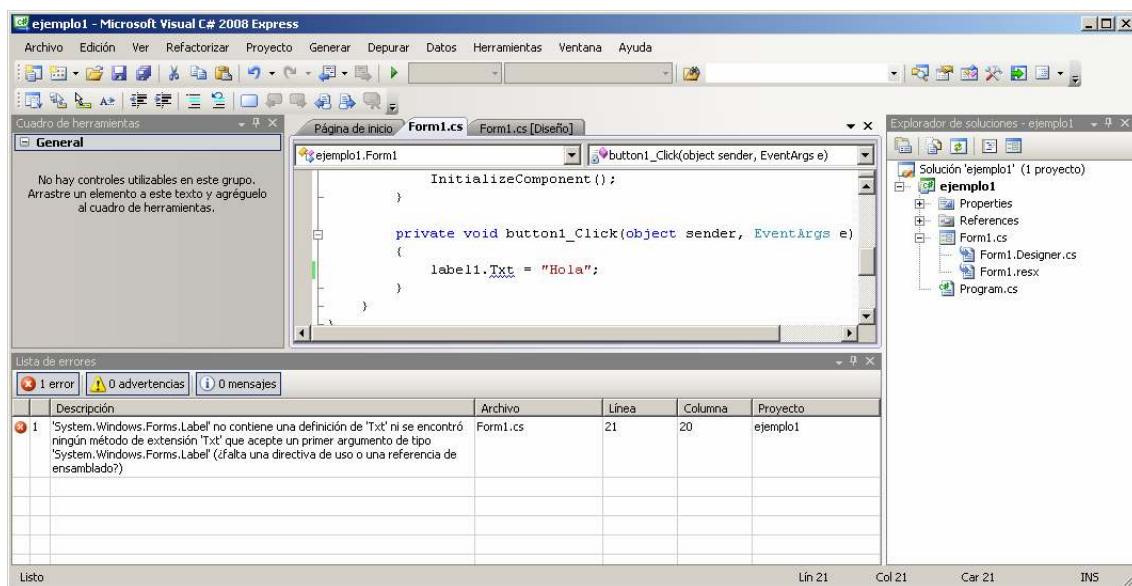
El "cuadro de herramientas", que nos permite colocar componentes visuales, como botones y casillas de texto, aparece "escondido" a la izquierda. Si hacemos un clic se despliega, y podemos fijarlo con el "pincho" que aparece en la esquina superior dercha, si no queremos que se vuelva a "esconder" automáticamente.



Podemos desplegar la categoría "componentes comunes" para acceder a los más habituales:



A partir de ahí, el manejo sería muy similar a lo que vimos de SharpDevelop en el apartado 8.2, con la diferencia de que no es necesario compilar para ver los errores, sino que se nos van indicando "al vuelo", a medida que tecleamos:

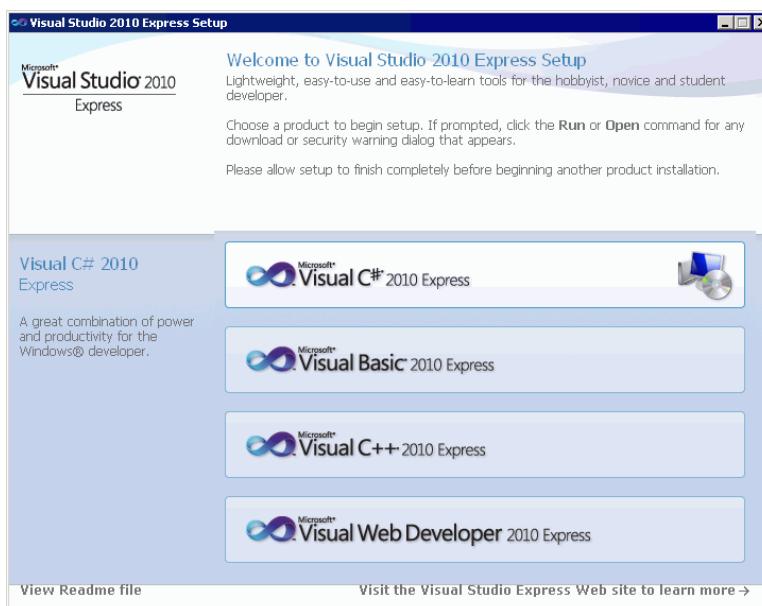


## Ap4.2. Visual Studio 2010 Express

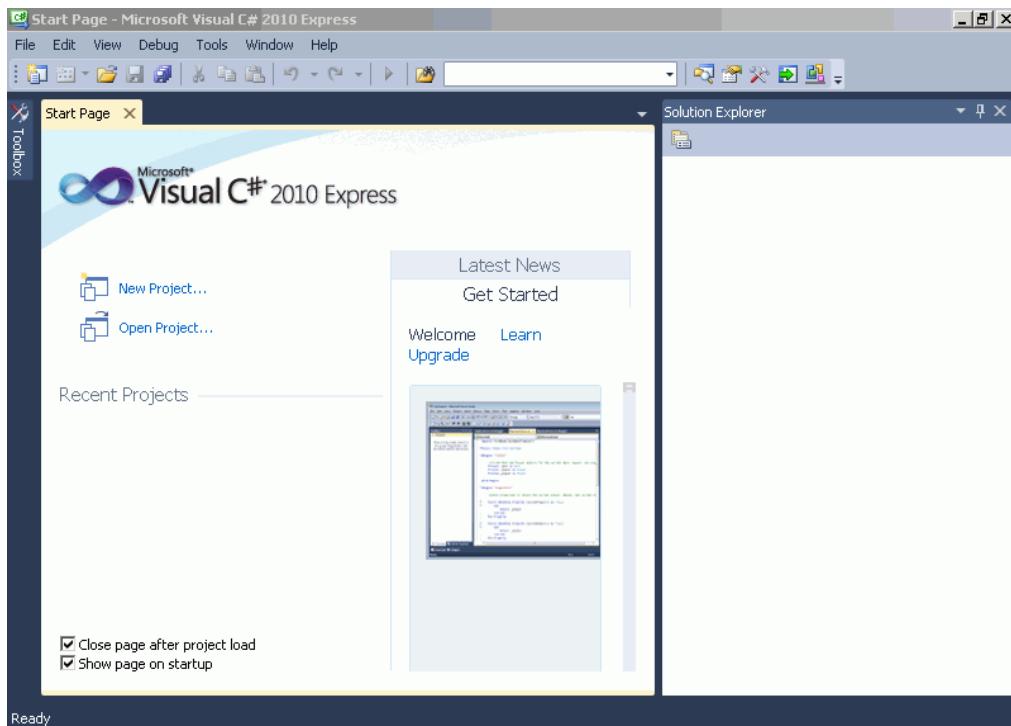
También está ya disponible Visual Studio 2010 (por ahora sólo en inglés), que se puede descargar de:

<http://www.microsoft.com/express/downloads/#2010-All>

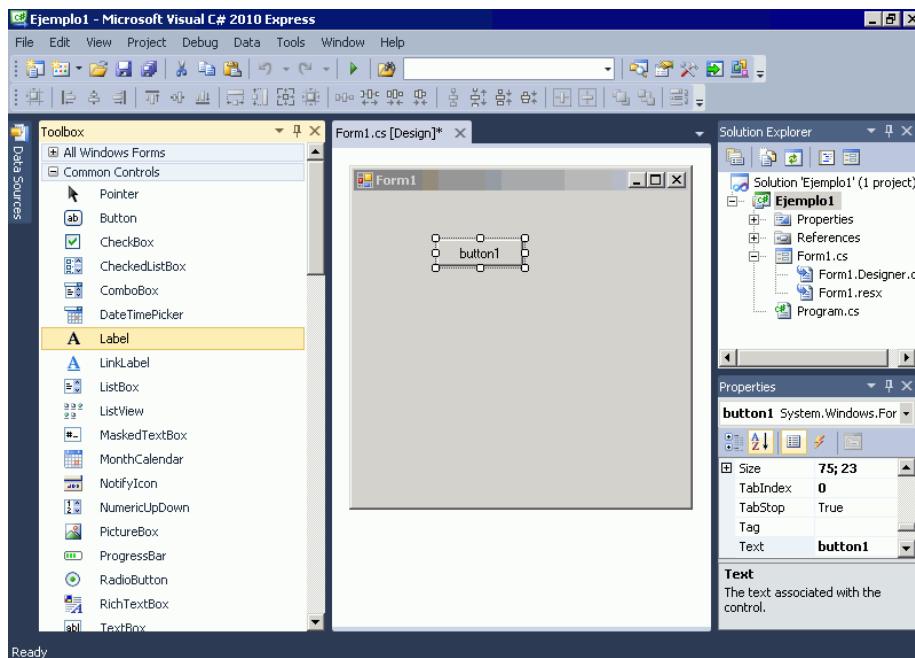
La instalación y el manejo son muy similares a los de la versión 2008, salvo por el detalle de que está en inglés y de que la nueva estética juega con tonos blancos y azules. Esta es la pantalla de instalación:



Ésta es la de inicio de Visual C#:



Y esta sería la ventana de trabajo, con nuestro formulario, la vista de clases a la derecha (solution explorer), el cuadro de herramientas a la izquierda (toolbox), y las propiedades de cada componente (properties) en la parte inferior derecha:



Revisões de este texto hasta la fecha:

- 0.97, de 01-feb-10. Corregida alguna errata en referencias del tema 2, que hablaban de ejemplos cuya numeración había cambiado. Incluido el ejemplo 14b, sobre el uso de "switch". Ampliado el apartado 6, para hablar de Get y Set, de constructores que se basan en otros, y de la palabra "this".
- 0.96, de 03-oct-10. Intercambiados los temas 2 y 3, para que no haya mucha carga teórica al principio. Ligeramente ampliado algún apartado de estos temas. Añadidos 4 apartados sobre SDL (10.11.3 a 10.11.6). Completado el apartado sobre expresiones regulares, al que faltaba un párrafo. La lista de cambios entre versiones (este apartado) pasa al final del texto.
- 0.95, de 01-jun-10. Recolocados (en distinto orden) los temas 8, 9, 10, para que el apartado sobre "otras bibliotecas" pase a ser el último de esos tres. Añadido el apartado 10.12 sobre servicios de red (cómo descargar una página web y comunicar dos ordenadores). Ampliado el apartado 9.6 sobre expresiones regulares e incluido un segundo ejemplo de acceso a bases de datos con SQLite en 10.10. Ampliado el apartado 2.2.3 para ver cómo convertir un número a binario o hexadecimal. Añadidos tres ejercicios propuestos al apartado 4.1 y otros dos al 4.4.3
- 0.94, de 07-may-10. Algunas correcciones menores en algunos apartados (por ejemplo, 6.3). Añadido el apartado 8.13 sobre Tao.SDL y el apéndice 4 sobre la instalación y uso básico de Visual Studio (en sus versiones 2008 Express y 2010 Express).
- 0.93, de 10-mar-10. Algunas correcciones menores en algunos apartados (por ejemplo, 2.1, 2.2, 3.1.2, 3.1.9, 4.1.4). Añadido el apartado 6.13 sobre SharpDevelop y cómo crear programas a partir de varios fuentes.
- 0.92, de 22-nov-09. Ampliado el tema 4 con un apartado sobre comparaciones de cadenas y otro sobre métodos de ordenación. Añadidos ejercicios propuestos en los apartados 4.4.8, 5.5 (2), 5.7 (2), 5.9.1.
- 0.91, de 19-nov-09, que incluye unos 30 ejercicios propuestos adicionales (en los apartados 1.2, 1.4, 1.8, 3.1.4, 3.1.5, 3.1.9, 3.2.1, 3.2.2, 3.2.3, 4.1.1, 4.1.2, 4.3.2, 4.5 y 5.10) y algunos apartados con su contenido ampliado (como el 4.4.6 y el 8.1).
- 0.90, de 24-may-09, como texto de apoyo para los alumnos de Programación en Lenguajes Estructurados en el I.E.S. San Vicente. (Primera versión completa; ha habido 4 versiones previas, incompletas, de distribución muy limitada).

# Índice alfabético

|                                   |                        |
|-----------------------------------|------------------------|
| <b>r</b>                          | <b>a</b>               |
| -, 19                             | @, 63, 246             |
| <b>[</b>                          | <b>l</b>               |
| --, 55                            | [,] (arrays), 71       |
| <b>!</b>                          | [] (arrays), 66        |
| !, 31                             | <b>\</b>               |
| !=, 28                            | \, 62                  |
| <b>#</b>                          | <b>^</b>               |
| #, 58                             | ^, 191                 |
| <b>%</b>                          | <b>{</b>               |
| %, 19                             | { y }, 11, 27          |
| %=, 56                            | <b> </b>               |
| <b>&amp;</b>                      | , 191                  |
| &, 191                            | , 31                   |
| & (dirección de una variable, 182 | <b>~</b>               |
| &&, 31                            | ~, 125, 191            |
| <b>*</b>                          | <b>+</b>               |
| *, 19                             | +, 19                  |
| * (punteros), 182                 | ++, 42, 55             |
| *=, 56                            | +=, 56                 |
| <b>,</b>                          | <b>&lt;</b>            |
| ,, 199                            | <, 28                  |
| .                                 | <<, 191                |
| .Net, 8                           | <b>=</b>               |
| <b>/</b>                          | <b>/</b>               |
| /, 19                             | =, 20                  |
| /*, 23                            | -=, 56                 |
| /**, 245                          | ==, 28                 |
| //, 23                            | <b>&gt;</b>            |
| ///, 245                          | >, 28                  |
| /=, 56                            | >>, 191                |
| <b>:</b>                          | <b>A</b>               |
| : (goto), 49                      | Abs, 105               |
| <b>?</b>                          | Acceso aleatorio, 154  |
| ?, 34                             | Acceso secuencial, 154 |
|                                   | Acos, 105              |
|                                   | Add (ArrayList), 172   |
|                                   | al azar, números, 104  |

Aleatorio, acceso, 154  
 aleatorios, números, 104  
 algoritmo, 9  
 alto nivel, 6  
 and, 191  
 Añadir a un fichero, 148  
 apilar, 170  
 Aplicación Windows, 203  
 Append, 162  
 AppendText, 148  
 Arco coseno, 105  
 Arco seno, 105  
 Arco tangente, 105  
 args, 109  
 Argumentos de un programa, 108  
 Aritmética de punteros, 185  
 array, 66  
 Array de objetos, 128  
 ArrayList, 172  
 Arrays bidimensionales, 70  
 Arrays de struct, 74  
 arreglo, 66  
 ASCII, 250  
 Asignación de valores, 20  
 Asignación en un "if", 31  
 asignaciones múltiples, 56  
 Asin, 105  
 Atan, 105  
 Atan2, 105  
 atributo, 122  
 azar, 104

**B**

BackgroundColor, 201  
**bajo nivel**, 7  
 base, 136  
 Base numérica, 59  
 Bases de datos, 217  
 BaseStream, 161  
 BASIC, 6  
 Begin, 156  
 Binario, 59, 251, 258, 264  
 BinaryReader, 154  
 BinaryWriter, 160  
 Bloque de programa, 44  
 BMP, 162  
 bool, 64  
 Booleanos, 64  
 borrar la pantalla, 201  
 break, 36, 46  
 bucle de juego, 226  
 bucle sin fin, 43  
 Bucle, 39  
 Bucle anidados, 43  
 bug, 236  
 burbuja, 88  
 byte, 54, 247

**C**

C, 6  
 C#, 7  
 Cadena modificable, 82  
 Cadenas de caracteres, 76  
 Cadenas de texto, 63  
 Carácter, 60

carpetas, 149  
 case, 36  
 caso contrario, 29  
 catch, 150

**Ch**

char, 36, 60

**C**

cifrar mensajes, 192  
 Cifras significativas, 57  
 clase, 112  
 class, 11, 112  
 Clear, 201  
 Close, 146  
 código máquina, 6  
 Códigos de formato, 58  
 cola, 171  
 Color de texto, 201  
 Coma fija, 56  
 Coma flotante, 56  
 Comentarios  
   recomendaciones, 242  
 Comillas (escribir), 62  
 CommandLine, 217  
 Comparación de cadenas, 81  
 CompareTo, 81  
 compiladores, 8  
 Compilar con mono, 17  
 Complemento, 191  
 Complemento a 1, 256  
 Complemento a 2, 256  
 Consola, 201  
 Console, 11, 201  
 ConsoleKeyInfo, 202  
 constantes, 192  
 constructor, 123  
 Contains, 171, 176, 177  
 ContainsKey, 177  
 ContainsValue, 176  
 continue, 47  
 Convert, 54  
 Convert.ToInt32, 24  
 Convertir a binario, 59  
 Convertir a hexadecimal, 59  
 Cos, 105  
 Coseno, 105  
 Coseno hiperbólico, 105  
 Cosh, 105  
 Create, 158  
 CreateDirectory, 215  
 CreateNew, 162  
 CreateText, 145  
 Current, 156

**D**

DateTime, 104, 213  
 Day, 213  
 debug, 236  
 decimal, 57  
 Decimal (sistema de numeración), 251  
 Declarar una variable, 20  
 Decremento, 55  
 default, 36

Depuración, 236  
 Dequeue, 171  
 Desbordamiento, 19  
 Descomposición modular, 93  
 Desplazamiento a la derecha, 191  
 Desplazamiento a la izquierda, 191  
 destructor, 125  
 diagrama de clases, 121  
 Diagramas de Chapin, 50  
 Diagramas de flujo, 32  
 Dibujo con Windows Forms, 212  
 Dinámica, memoria, 169  
 dirección de memoria, 182  
 Directorios, 215  
 DirectoryInfo, 215  
 Diseño modular de programas, 93  
 Distinto de, 28  
 División, 19  
 do ... while, 40  
 Doble precisión, 57  
 Documentación, 241  
 Dot Net Framework, 8  
 double, 57  
 Doxygen, 245  
 DrawLine, 213

**E**

E, 106  
 ejecutable, 8  
 elevado a, 105  
 else, 29  
 Encapsulación, 111  
 End, 157  
 Enqueue, 171  
 ensamblador, 7  
 ensambladores, 8  
 entero, 54  
 enteros negativos, 255  
 Entorno, 216  
 enum, 192  
 Enumeraciones, 192  
 enumeradores, 178  
 Environment, 217  
 Environment.Exit, 109  
 ERRORLEVEL, 109  
 escritura indentada, 28  
 Espacios de nombres, 189  
 Estructuras, 73  
 Estructuras alternativas, 26  
 Estructuras anidadas, 75  
 Estructuras de control, 26  
 Estructuras dinámicas, 169  
 Estructuras repetitivas, 39  
 Euclides, 108  
 Excepciones, 150  
 EXE, 8  
 Exists (ficheros), 149  
 Exp, 105  
 Exponencial, 105  
 Expresiones regulares, 196

**F**

factorial, 106  
 false, 64  
 falso, 64

Fecha y hora, 213  
 Fibonacci, 108  
 Fichero físico, 153  
 Fichero lógico, 153  
 Ficheros, 145  
 Ficheros binarios, 154  
 Ficheros de texto, 145  
 Ficheros en directorio, 216  
 FIFO, 171  
 File.Exists, 149  
 FileAccess.ReadWrite, 167  
 FileMode.Open, 155  
 FileStream, 155  
 fin de fichero, 147  
 fixed, 187  
 float, 57  
 for, 42  
 foreach, 83  
 ForegroundColor, 201  
 Formatear números, 58  
 fuente, 10  
 función de dispersión, 176  
 Funciones, 93  
 Funciones matemáticas, 105  
 Funciones virtuales, 132

**G**

get, 194  
 Get, 114  
 GetEnumerator, 178  
 GetFiles, 216  
 GetKey, 175  
 gigabyte, 247  
 gmcs, 202  
 goto, 49  
 goto case, 36

**H**

Herencia, 111  
 Hexadecimal, 59  
 Hora, 213

**I**

Identificadores, 22  
 if, 26  
 Igual a, 28  
 Incremento, 55  
 IndexOf, 78  
 IndexOfKey, 175  
 inseguro (bloque "unsafe"), 182  
 Inserción directa, 89  
 Insert, 79, 172  
 instante actual:, 104  
 int, 20, 54  
 intérprete, 8  
 interrumpir un programa, 109  
 Introducción de datos, 24  
 IOException, 162  
 iterativa, 108

**J**

JavaDoc, 246

Juegos, 219

## K

Key, 202  
KeyAvailable, 201  
KeyChar, 202  
kilobyte, 247

## L

LastIndexOf, 78  
Lectura y escritura en un mismo fichero, 166  
Length (cadenas), 77  
Length (fichero), 157  
lenguaje C, 6  
lenguaje máquina, 6  
LIFO, 170  
línea de comandos, 108  
Líneas, dibujar, 212  
lista, 172

## Li

llaves, 11

## L

Log, 105  
Log10, 105  
Logaritmo, 105  
long, 54  
Longitud de una cadena, 77

## M

Main, 11  
máquina virtual, 8  
matemáticas, funciones, 105  
Mayor que, 28  
mayúsculas, 78  
mcs, 17  
megabyte, 247  
Memoria dinámica, 169  
Menor que, 28  
MessageBox, 207  
métodos, 122  
Microsoft, 258  
Mientras (condición repetitiva), 39  
Millisecond, 104  
Modificar parámetros, 101  
Módulo (resto de división), 19  
Mono, 10, 12  
Month, 213  
Mostrar el valor de una variable, 21  
MoveNext, 179  
Multiplicación, 19

## N

n, 61  
namespace, 189  
Negación, 19  
Net, 8  
new (arrays), 66  
new (redefinir métodos), 118

nibble, 252  
No, 31  
not, 191  
Notepad++, 18  
Now, 213  
null (fin de fichero), 147  
Números aleatorios, 104  
números enteros, 18  
Números reales, 56

## O

0, 31  
Objetos, 111  
octal, 252  
ocultación de datos, 114  
OpenOrCreate, 162  
OpenRead, 155  
OpenText, 146  
OpenWrite, 158  
Operaciones abreviadas, 56  
Operaciones aritméticas, 19  
Operaciones con bits, 191  
operador coma, 199  
Operador condicional, 34  
Operadores lógicos, 31  
Operadores relacionales, 28  
or, 191  
Ordenaciones simples, 88  
OSVersion, 217  
out, 196  
override, 132

## P

Parámetros de Main, 108  
Parámetros de salida, 196  
Parámetros de una función, 95  
parámetros por referencia, 102  
parámetros por valor, 102  
Pascal, 6  
Paso a paso (depuración), 236  
pausa, 215  
Peek, 171  
Pi, 106  
pila, 170  
Polimorfismo, 111, 126  
POO, 111  
pop, 170  
Posición del cursor, 201  
Posición en el fichero, 156  
postdecremento, 55  
postincremento, 55  
Potencia, 105  
Pow, 105  
Precedencia de los operadores, 19  
predecremento, 55  
preincremento, 55  
Prioridad de los operadores, 19  
private, 118  
Process.Start, 216  
Producto lógico, 191  
programa, 6  
Programación orientada a objetos, 111  
Propiedades, 194  
protected, 118  
Prueba de programas, 239

Pseudocódigo, 9  
 public, 11, 117  
 public (struct), 73  
 Punteros, 169, 182  
 Punto Net, 8  
 Puntos de ruptura, 238  
 push, 170

**Q**

Queue, 171

**R**

Raíz cuadrada, 106  
 Random, 104  
 Rango de valores (enteros), 54  
 Read (FileStream), 155  
 ReadByte, 155, 156  
 Readkey, 201  
 ReadLine, 24  
 ReadLine (fichero), 146  
 ReadString, 155  
 real (tipo de datos), 56  
 recolector de basura, 187  
 Recursividad, 106  
 Redondear un número, 105  
 ref, 102, 196  
 Registro, 153  
 Registros, 73  
 Remove, 79  
 Replace, 79  
 Reserva de espacio, 185  
 Resta, 19  
 Resto de la división, 19  
 return, 96  
 Round, 105  
 rutas, 149

**S**

sangrado, 28  
 sbyte, 54  
 SDL, 219  
 secuencial, 154  
 Secuencias de escape, 61  
 Seek, 156  
 SeekOrigin, 156  
 Selección directa, 88  
 Seno, 106  
 Sentencias compuestas, 27  
 set, 194  
 Set, 114  
 SetByIndex, 176  
 SetCursorPosition, 201  
 SharpDevelop, 140, 203  
 short, 54  
 si no, 29  
 Signo y magnitud, 256  
 Simple precisión, 57  
 Sin, 106  
 Sistema binario, 251, 258, 264  
 Sistema de numeración, 59  
 Sistema decimal, 251  
 Sleep, 215  
 Sobrecarga, 126

Sobrecarga de operadores, 139  
 Sort, 172  
 SortedList, 175  
 Split, 80  
 SQLite, 217  
 Sqrt, 106  
 Stack, 170  
 stackalloc, 185  
 static, 122  
 StreamReader, 146  
 StreamWriter, 145  
 string, 38, 63  
 String.Compare, 81  
 StringBuilder, 81, 82  
 struct, 73  
 struct anidados, 75  
 Subcadena, 78  
 Substring, 78  
 Suma, 19  
 Suma exclusiva, 191  
 Suma lógica, 191  
 switch, 35  
 System, 11  
 System.Drawing, 212

**T**

tabla, 66  
 Tablas bidimensionales, 70  
 tablas hash, 176  
 Tan, 106  
 Tangente, 106  
 Tanh, 106  
 Teclado, 201  
 Temporización, 213  
 terabyte, 247  
 this, 138  
 Thread, 215  
 Thread.Sleep, 215  
 Tipo de datos carácter, 60  
 Tipos de datos enteros, 54  
 Title, 201  
 ToInt32, 24, 54  
 ToLower, 78  
 ToString, 58  
 ToUpper, 78  
 true, 64  
 Truncate, 162  
 try, 150

**U**

uint, 54  
 ulong, 54  
 UML, 121  
 Unicode, 250  
 unsafe, 182  
 ushort, 54  
 using, 24

**V**

Valor absoluto, 105  
 Valor devuelto por una función, 96  
 Variables, 20  
 Variables globales, 98

Variables locales, 98  
vector, 66  
verdadero, 64  
virtual, 132  
Visibilidad (POO), 117, 138  
Visual C#, 258  
Visual Studio, 258  
void, 93

## W

WaitForExit, 216  
while, 39  
Windows Forms, 203  
Write, 35

Write (BinaryWriter), 160  
Write (FileStream), 158  
WriteByte, 158  
WriteLine, 11  
WriteLine (ficheros), 145

## X

xor, 191

## Y

Y, 31  
Year, 213