

Introducción a la programación con C#

Este texto es una introducción a la programación de ordenadores, usando el lenguaje C#.

Está organizado de una forma ligeramente distinta a los libros de texto "convencionales", procurando incluir ejercicios prácticos lo antes posible, para evitar que un exceso de teoría en los primeros temas haga el texto pesado de seguir.

Este texto ha sido escrito por Nacho Cabanes. Si quiere conseguir la última versión, estará en mi página web:

www.nachocabanes.com

Este texto es de **libre distribución** ("gratis"). Se puede distribuir a otras personas libremente, siempre y cuando no se modifique. Si le gustan los formalismos, esto sale a equivaler a una licencia **Creative Commons** BY-NC-ND: reconocimiento del autor, no se puede hacer uso comercial (no se puede "vender" este curso), no se puede crear obras derivadas.

Este texto se distribuye "tal cual", sin garantía de ningún tipo, implícita ni explícita. Aun así, mi intención es que resulte útil, así que le rogaría que me comunique cualquier error que encuentre.

Para cualquier sugerencia, no dude en contactar conmigo a través de mi web.

Revisión actual: 0.99zz

Contenido

Contenido	2
0. Conceptos básicos sobre programación	9
0.1. Programa y lenguaje	9
0.2. Lenguajes de alto nivel y de bajo nivel	9
0.3. Ensambladores, compiladores e intérpretes	11
0.4. Pseudocódigo	14
1. Toma de contacto con C#	16
1.1. ¿Qué es C #? ¿Qué entorno usaremos?	16
1.2. Escribir un texto en C#	17
1.3. Cómo probar este programa	19
1.3.1. Cómo probarlo con Mono en Linux	19
1.3.2. Cómo probarlo con Mono en Windows	24
1.3.3. Otros editores más avanzados para Windows	33
1.4. Mostrar números enteros en pantalla	34
1.5. Operaciones aritméticas básicas	34
1.5.1. Operadores	34
1.5.2. Orden de prioridad de los operadores	35
1.5.3. Introducción a los problemas de desbordamiento	36
1.6. Introducción a las variables: int	36
1.6.1. Definición de variables: números enteros	36
1.6.2. Asignación de valores	37
1.6.3. Mostrar el valor de una variable en pantalla	38
1.7. Identificadores	40
1.8. Comentarios	41
1.9. Datos por el usuario: ReadLine	43
1.10. using System	44
1.11. Escribir sin avanzar de línea	45
2. Estructuras de control	47
2.1. Estructuras alternativas	47
2.1.1. if	47
2.1.2. if y sentencias compuestas	48
2.1.3. Operadores relacionales: <, <=, >, >=, ==, !=	49
2.1.4. if-else	50
2.1.5. Operadores lógicos: &&, , !	52
2.1.6. El peligro de la asignación en un "if"	54
2.1.7. Introducción a los diagramas de flujo	55
2.1.8. Operador condicional: ?	58
2.1.9. switch	60
2.2. Estructuras repetitivas	63
2.2.1. while	63
2.2.2. do ... while	65
2.2.3. for	67
2.2.4. Bucles sin fin	69

2.2.5. Bucles anidados	69
2.2.6. Repetir sentencias compuestas	70
2.2.7. Contar con letras	71
2.2.8. Declarar variables en un "for"	72
2.2.9. Las llaves son recomendables	74
2.2.10. Interrumpir un bucle: break	75
2.2.11. Forzar la siguiente iteración: continue	77
2.2.12. Equivalencia entre "for" y "while"	78
2.2.13. Ejercicios resueltos sobre bucles	80
2.3. Saltar a otro punto del programa: goto	82
2.4. Más sobre diagramas de flujo. Diagramas de Chapin	83
2.5. foreach	85
2.6. Recomendación de uso para los distintos tipos de bucle	86
2.7. Una alternativa para el control errores: las excepciones	88
3. Tipos de datos básicos	92
3.1. Tipo de datos entero	92
3.1.1. Tipos de datos para números enteros	92
3.1.2. Conversiones de cadena a entero	93
3.1.3. Incremento y decremento	94
3.1.4. Operaciones abreviadas: +=	96
3.1.5. Asignaciones múltiples	96
3.2. Tipo de datos real	97
3.2.1. Coma fija y coma flotante	97
3.2.2. Simple y doble precisión	98
3.2.3. Pedir números reales al usuario	100
3.2.4. Conversión de tipos (typecast)	102
3.2.5. Formatear números	103
3.2.6. Cambios de base	105
3.3. Tipo de datos carácter	107
3.3.1. Leer y mostrar caracteres	107
3.3.2. Secuencias de escape: \n y otras	108
3.4. Toma de contacto con las cadenas de texto	110
3.5. Los valores "booleanos"	111
4. Arrays, estructuras y cadenas de texto	113
4.1. Conceptos básicos sobre arrays o tablas	113
4.1.1. Definición de un array y acceso a los datos	113
4.1.2. Valor inicial de un array	114
4.1.3. Recorriendo los elementos de una tabla	115
4.1.4. Operaciones habituales con arrays: buscar, añadir, insertar, borrar	118
4.1.5. Constantes	121
4.2. Arrays bidimensionales	122
4.3. Estructuras o registros	125
4.3.1. Definición y acceso a los datos	125
4.3.2. Arrays de structs	126
4.3.3. structs anidados	127
4.4. Cadenas de caracteres	128
4.4.1. Definición. Lectura desde teclado	128
4.4.2. Cómo acceder a las letras que forman una cadena	130

4.4.3. Longitud de la cadena	130
4.4.4. Extraer una subcadena	131
4.4.5. Buscar en una cadena	132
4.4.6. Otras manipulaciones de cadenas	133
4.4.7. Descomponer una cadena en fragmentos	135
4.4.8. Comparación de cadenas	137
4.4.9. Una cadena modificable: StringBuilder	138
4.5. Recorriendo arrays y cadenas con "foreach"	139
4.6. Ejemplo completo	140
4.7. Ordenaciones simples	145
4.8. Otros editores más avanzados: Notepad++ y Geany	151
5. Introducción a las funciones	156
5.1. Diseño modular de programas: Descomposición modular	156
5.2. Conceptos básicos sobre funciones	156
5.3. Parámetros de una función	158
5.4. Valor devuelto por una función. El valor "void"	160
5.5. Variables locales y variables globales	163
5.6. Los conflictos de nombres en las variables	165
5.7. Modificando parámetros	167
5.8. El orden no importa	169
5.9. Algunas funciones útiles	170
5.9.1. Números aleatorios	170
5.9.2. Funciones matemáticas	171
5.9.3. Pero hay muchas más funciones...	174
5.10. Recursividad	174
5.11. Parámetros y valor de retorno de "Main"	177
6. Programación orientada a objetos	180
6.1. ¿Por qué los objetos?	180
6.2. Objetos y clases en C#	186
6.3. Proyectos a partir de varios fuentes	191
6.4. La herencia	205
6.5. Visibilidad	208
6.6. Constructores y destructores	212
6.7. Polimorfismo y sobrecarga	215
6.8. Orden de llamada de los constructores	216
7. Utilización avanzada de clases	219
7.1. La palabra "static"	219
7.2. Arrays de objetos	221
7.3. Funciones virtuales. La palabra "override"	225
7.4. Llamando a un método de la clase "padre"	229
7.5. La palabra "this": el objeto actual	232

7.6. Sobrecarga de operadores	237
7.7. Proyectos completos propuestos	239
8. Manejo de ficheros	242
8.1. Escritura en un fichero de texto	242
8.2. Lectura de un fichero de texto	244
8.3. Lectura hasta el final del fichero	245
8.4. Añadir a un fichero existente	247
8.5. Ficheros en otras carpetas	248
8.6. Saber si un fichero existe	249
8.7. Más comprobaciones de errores: excepciones	250
8.8. Conceptos básicos sobre ficheros	253
8.9. Leer un byte de un fichero binario	253
8.10. Leer hasta el final de un fichero binario	256
8.11. Leer bloques de datos de un fichero binario	258
8.12. La posición en el fichero	259
8.13. Leer datos nativos	261
8.14. Ejemplo completo: leer información de un fichero BMP	263
8.15. Escribir en un fichero binario	267
8.16. Leer y escribir en un mismo fichero binario	271
9. Persistencia de objetos	274
9.1. ¿Por qué la persistencia?	274
9.2. Creando un objeto "serializable"	275
9.3. Empleando clases auxiliares	277
9.4. Volcando a un fichero de texto	283
10. Acceso a bases de datos relacionales	289
10.1. Nociones mínimas de bases de datos relacionales	289
10.2. Nociones mínimas de lenguaje SQL	289
10.2.1. Creando la estructura	289
10.2.2. Introduciendo datos	290
10.2.3. Mostrando datos	291
10.3. Acceso a bases de datos con SQLite	292
10.4. Un poco más de SQL: varias tablas	297
10.4.1. La necesidad de varias tablas	297
10.4.2. Las claves primarias	298
10.4.3. Enlazar varias tablas usando SQL	298
10.4.4. Varias tablas con SQLite desde C#	301
10.5. Borrado y modificación de datos	303
10.6. Operaciones matemáticas con los datos	304
10.7. Grupos	305
10.8. Un ejemplo completo con C# y SQLite	306

10.9. Nociones mínimas de acceso desde un entorno gráfico	310
11. Punteros y gestión dinámica de memoria	311
11.1. ¿Por qué usar estructuras dinámicas?	311
11.2. Una pila en C#	312
11.3. Una cola en C#	314
11.4. Las listas	316
11.4.1. ArrayList	316
11.4.2. SortedList	319
11.5. Las "tablas hash"	320
11.6. Los "enumeradores"	322
11.7. Cómo "imitar" una pila usando "arrays"	324
11.8. Introducción a los "generics"	326
11.9. Los punteros en C#	329
11.9.1. ¿Qué es un puntero?	329
11.9.2. Zonas "inseguras": unsafe	330
11.9.3. Uso básico de punteros	331
11.9.4. Zonas inseguras	332
11.9.5. Reservar espacio: stackalloc	333
11.9.6. Aritmética de punteros	334
11.9.7. La palabra "fixed"	335
12. Algunas bibliotecas adicionales de uso frecuente	337
12.1. Fecha y hora. Temporización	337
12.2. Más posibilidades de la "consola"	340
12.3. Lectura de directorios	344
12.4. El entorno. Llamadas al sistema	347
12.5. Datos sobre "el entorno"	348
12.6. Algunos servicios de red.	349
13. Otras características avanzadas de C#	354
13.1. Espacios de nombres	354
13.2. Operaciones con bits	356
13.3. Enumeraciones	358
13.4. Propiedades	360
13.5. Introducción a las expresiones regulares.	361
13.6. El operador coma	364
13.7. Variables con tipo implícito	365
13.8. Contacto con LINQ	366
13.9. Lo que no vamos a ver...	368
14. Depuración, prueba y documentación de programas	369
14.1. Conceptos básicos sobre depuración	369
14.2. Depurando desde Visual Studio	369
14.3. Prueba de programas	372

14.4. Documentación básica de programas	377
14.4.1. Consejos para comentar el código	378
14.4.2. Generación de documentación a partir del código fuente.	381
Apéndice 1. Unidades de medida y sistemas de numeración	385
Ap1.1. bytes, kilobytes, megabytes...	385
Ap1.2. Unidades de medida empleadas en informática (2): los bits	386
Apéndice 2. El código ASCII	389
Apéndice 3. Sistemas de numeración.	391
Ap3.1. Sistema binario	391
Ap3.2. Sistema octal	393
Ap3.3. Sistema hexadecimal	395
Ap3.4. Representación interna de los enteros negativos	397
Apéndice 4. SDL	400
Ap4.1. Juegos con Tao.SDL	400
Ap4.2. Mostrar una imagen estática	400
Ap4.3. Una imagen que se mueve con el teclado	403
Ap4.4. Simplificando con clases auxiliares	405
Ap4.5. Un fuente más modular: el "bucle de juego"	409
Ap4.6. Escribir texto	411
Ap4.7. Colisiones simples	413
Ap4.8. Imágenes PNG y JPG	417
Ap4.9. ¿Por dónde seguir?	418
Apéndice 5. Contacto con los entornos gráficos	420
Ap5.1. Creación de formularios, botones y etiquetas	420
Ap5.2. Cambios de apariencia. Casillas de texto para sumar dos números	424
Ap5.3. Usando ventanas predefinidas	428
Ap5.4. Una aplicación con dos ventanas	430
Ap5.5. Otros componentes visuales	433
Ap5.6. Dibujando con Windows Forms	434
Revisiones de este texto	437
Índice alfabético	439

0. Conceptos básicos sobre programación

0.1. Programa y lenguaje

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.

0.2. 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 y detallar ciertas cosas como el nombre del programa o dónde empieza y termina éste, 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, porque los programas en C suelen necesitar incluir bibliotecas externas y devolver códigos de error (incluso cuando todo ha ido bien):

```
#include <stdio.h>

int main()
{
    printf("Hola");
    return 0;
}
```

En **C#** hay que dar todavía más pasos para conseguir lo mismo, porque, como veremos, cada programa será "una clase":

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

Como se puede observar, a medida que los lenguajes evolucionan, son capaces de ayudar al programador en más tareas, pero a la vez, los programas sencillos se vuelven más complicados. Afortunadamente, no todos los lenguajes siguen esta regla, y algunos se han diseñado de forma que las tareas simples sean (de nuevo) sencillas de programar. Por ejemplo, para escribir algo en pantalla usando el lenguaje **Python** haríamos:

```
print("Hello")
```

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
saludo db 'Hola',0dh,0ah,'$'

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

    mov     ah,9
    mov     dx,offset saludo
    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 ciertos errores).

0.3. Ensambladores, compiladores e intérpretes

Como hemos visto, las órdenes que nosotros hemos escrito (lo que se conoce como "programa **fuentes**") deben convertirse a lo que el ordenador comprende (obteniendo un "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 dicho ordenador no tenga un compilador de Pascal instalado. Eso sí, no funcionaría en otro ordenador que tuviera un sistema operativo distinto (por ejemplo, Linux o Mac OS X).

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 que hemos escrito en lenguaje de alto nivel a su equivalente en código máquina, orden por orden, justo en el momento en que hay que procesar cada una de las órdenes.

Los intérpretes siguen siendo muy frecuentes hoy en día. Por ejemplo, en un servidor web es habitual crear programas usando lenguajes como PHP, ASP o Python, y que estos programas no se conviertan a un ejecutable, sino que sean analizados y puestos en funcionamiento en el momento en el que se solicita la correspondiente página web.

Actualmente existe una alternativa más, 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 "**plataforma punto net**"): los programas que creemos con herramientas como Visual Studio serán unos ejecutables que funcionarán en cualquier ordenador que tenga instalada dicha "plataforma .Net", algo que suele ocurrir en las versiones recientes de Windows y que se puede conseguir de forma un poco más artesanal en plataformas Linux y Mac, gracias a un "clon" de la "plataforma .Net" que es de libre distribución, conocido como "proyecto Mono".

Ejercicios propuestos

- **(0.3.1)** Localiza en Internet el intérprete de BASIC llamado Bywater Basic, en su versión para el sistema operativo que estés utilizando y prueba el primer programa de ejemplo que se ha visto en el apartado 0.1. También puedes usar cualquier "ordenador clásico" (de principios de los años 80) y otros muchos BASIC modernos, como Basic256. (**Nota:** no es necesario realizar este ejercicio para seguir adelante con el curso; puedes omitirlo si no encuentras ningún intérprete de BASIC).
- **(0.3.2)** Localiza en Internet el compilador de Pascal llamado Free Pascal, en su versión para el sistema operativo que estés utilizando, instálalo y prueba el segundo programa de ejemplo que se ha visto en el apartado 0.1. (**Nota:** no es necesario realizar este ejercicio para seguir adelante con el curso; puedes omitirlo si no quieres instalar en tu ordenador software que luego no vayas a seguir utilizando).
- **(0.3.3)** Localiza un compilador de C para el sistema operativo que estés utilizando (si es Linux o alguna otra versión de Unix, es fácil que se encuentre ya instalado) y prueba el tercer programa de ejemplo que se ha visto en el apartado 0.1. (**Nota:** no es necesario realizar este ejercicio para seguir adelante con el curso; puedes omitirlo si no quieres instalar en tu equipo software que después quizá no vuelvas a utilizar).
- **(0.3.4)** Descarga un intérprete de Python (ya estará preinstalado si usas Linux) o busca en Internet "try Python web" para probarlo desde tu navegador web, y prueba el quinto programa de ejemplo que se ha visto en

el apartado 0.1. (**Nota:** nuevamente, no es necesario realizar este ejercicio para seguir adelante con el curso).

0.4. Pseudocódigo

A pesar de que los lenguajes de alto nivel se acercan al lenguaje natural (inglés), que nosotros empleamos, es habitual no usar ningún lenguaje de programación concreto cuando queremos plantear inicialmente los pasos necesarios para resolver un problema, sino emplear un lenguaje de programación ficticio, no tan estricto, muchas veces 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 es algo un poco más estricto que eso: por ejemplo, un algoritmo debe estar formado por un número finito de pasos. Por tanto, un programa de ordenador es un algoritmo expresado usando 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
```

Como se ve en este ejemplo, el pseudocódigo suele ser menos detallado que un lenguaje de programación "real" y expresar las acciones de forma más general, buscando concretar las ideas más que la forma real de llevarlas a cabo. Por ejemplo, ese "conectar con central de cobros" correspondería a varias órdenes individuales en cualquier lenguaje de programación.

Ejercicios propuestos

- **(0.4.1)** Localiza en Internet el intérprete de Pseudocódigo llamado PseInt y prueba escribir el siguiente programa. (**Nota:** no es necesario realizar este

ejercicio para seguir adelante con el curso; puedes omitirlo si no te apetece instalar en tu equipo software que luego no vayas a seguir utilizando).

```
Proceso EjemploDeSuma
    Escribir 2+3
FinProceso
```

1. Toma de contacto con C#

1.1. ¿Qué es C #? ¿Qué entorno usaremos?

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 y es más difícil cometer errores.

Se trata de un lenguaje creado por Microsoft para realizar programas para su plataforma .NET, pero fue estandarizado posteriormente por ECMA y por ISO, y 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, junto con un editor de texto para programadores. 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). Existen otros entornos alternativos, como SharpDevelop o MonoDevelop, que también comentaremos.

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

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

Tras el siguiente apartado veremos un ejemplo de entorno desde el que realizar nuestros programas, dónde localizarlo y cómo instalarlo.

1.2. 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 analizarlo ahora con más detalle:

```
public class Ejemplo_01_02a
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

Esto escribe "Hola" en la pantalla. Pero hay muchas "cosas raras" alrededor de ese "Hola", de modo vamos a comentarlas antes de proseguir, aunque muchos de los detalles los aplazaremos para más adelante. En este primer análisis, iremos desde 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");` : `WriteLine` siempre irá precedido de "Console." porque es una orden de manejo de la "consola" (la pantalla "negra" en modo texto del sistema operativo).
- `System.Console.WriteLine("Hola");` : 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 (Main).
- `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` Ejemplo_01_02a : De momento pensaremos que "Ejemplo_01_02a" es el nombre de nuestro programa. Una línea como esa deberá existir también siempre en nuestros programas (aunque el nombre no tiene por qué ser tan "rebuscado"), 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 (1.2.1): Crea un programa en C# que te salude por tu nombre (por ejemplo, "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 donde termina una orden y donde empieza la siguiente son los puntos y coma y las llaves. Por ese motivo, el programa anterior se podría haber escrito también así (aunque no es aconsejable, porque puede resultar menos legible):

```
public class Ejemplo_01_02b {
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 Ejemplo_01_02c {
    public static void Main(){
        System.Console.WriteLine("Hola");
    }
}
```

```

    }
}
```

(esta es la forma que se empleará preferentemente en este texto cuando estemos trabajando con fuentes de mayor tamaño, para que ocupe un poco menos de espacio; en los primeros fuentes usaremos el "estilo C", que tiende a resultar más legible).

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.

1.3. Cómo probar este programa

1.3.1. Cómo probarlo con Mono en Linux

Para alguien acostumbrado a sistemas como Windows o Mac OS, hablar de Linux puede sonar a que se trata de algo apto sólo para expertos. Eso no necesariamente es así, y, de hecho, para un aprendiz de programador puede resultar justo al contrario, porque Linux tiene compiladores e intérpretes de varios lenguajes ya preinstalados, y otros son fáciles de instalar en unos pocos clics.

La instalación de Linux, que podría tener la dificultad de crear particiones para coexistir con nuestro sistema operativo habitual, hoy en día puede realizarse de forma simple, usando software de virtualización gratuito, como VirtualBox, que permite tener un "ordenador virtual" dentro del nuestro, e instalar Linux en ese "ordenador virtual" sin interferir con nuestro sistema operativo habitual.

Así, podemos instalar VirtualBox, descargar la imagen ISO del CD o DVD de instalación de algún Linux que sea reciente y razonablemente amigable, arrancar VirtualBox, crear una nueva máquina virtual y "cargar esa imagen de CD" para instalar Linux en esa máquina virtual.

En este caso, yo comentaré los pasos necesarios para usar Linux Mint (en su versión 17 Cinnamon) como entorno de desarrollo:

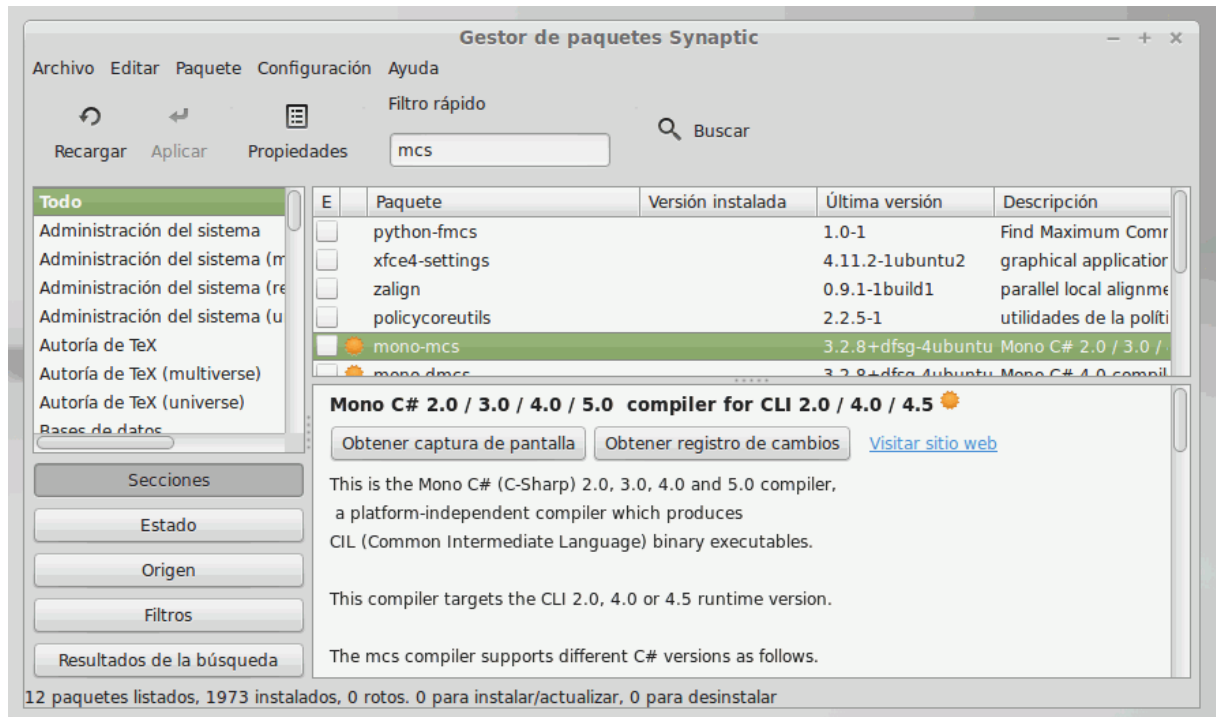


En primer lugar, deberemos entrar al instalador de software de nuestro sistema, que para las versiones de Linux Mint basadas en escritorios derivados de Gnome (como es el caso de Mint 17 Cinnamon), suele ser un tal "Gestor de paquetes Synaptic":

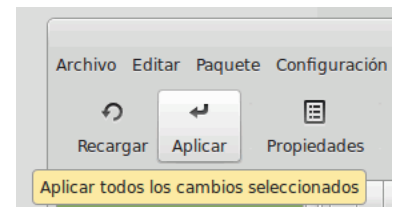


Se nos pedirá nuestra contraseña de usuario (la que hayamos utilizado en el momento de instalar Linux), y aparecerá la pantalla principal de Synaptic, con una enorme lista de software que podemos instalar. En esta lista, aparece una casilla

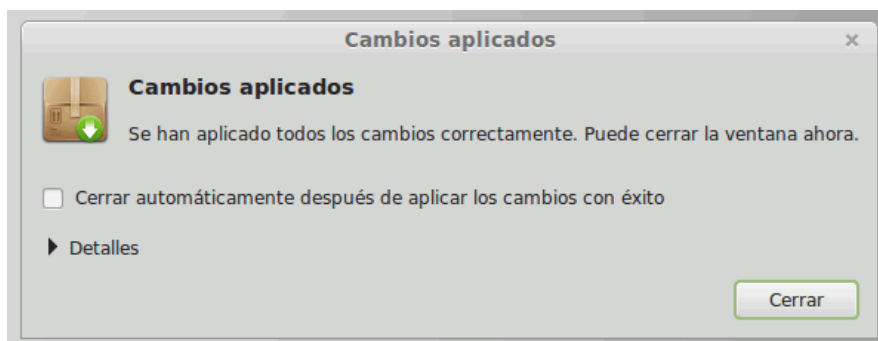
de texto llamada "Filtro rápido", en la que podemos teclear "mcs" para que nos aparezca directamente nuestro compilador Mono:



Entre otros paquetes, posiblemente veremos uno llamado "mono-mcs", en cuya descripción se nos dirá que es el "Mono C# Compiler". Al hacer doble clic se nos avisará en el caso (habitual) de que haya que instalar algún otro paquete adicional y entonces ya podremos pulsar el botón "Aplicar":



Se descargarán los ficheros necesarios, se instalarán y al cabo de un instante se nos avisará de que se han aplicado todos los cambios:

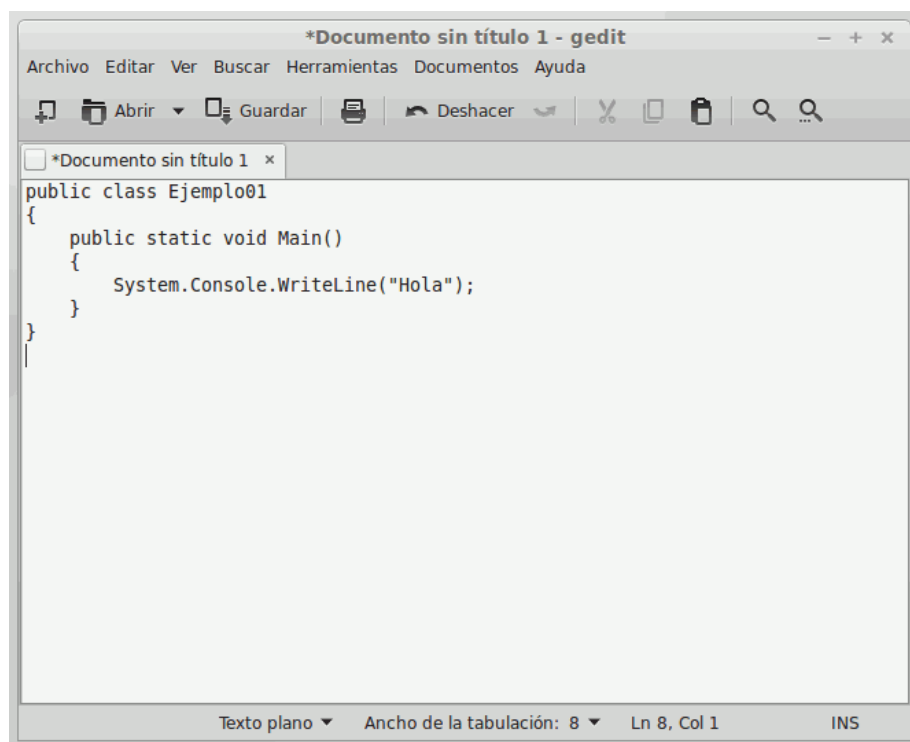


Ya tenemos instalado el compilador, que convertirá nuestros programas en algo que el ordenador entienda. Para teclear nuestros programas necesitaremos un editor de texto, pero eso es algo que viene preinstalado en cualquier Linux. Por

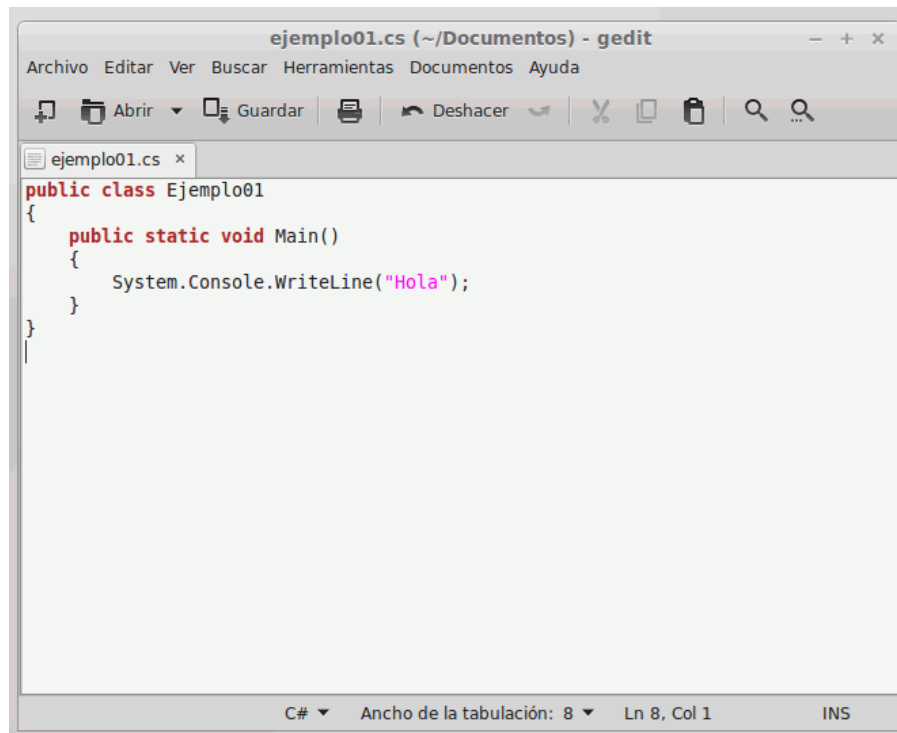
ejemplo, en esta versión de Linux encontraremos un editor de textos llamado "gedit" dentro del apartado de accesorios:



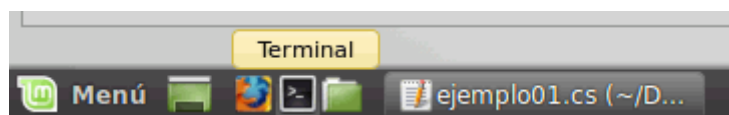
En este editor podemos teclear nuestro programa, que inicialmente se verá con letras negras sobre fondo blanco:



Cuando lo guardemos con un nombre terminado en ".cs" (como "ejemplo01.cs"), el editor sabrá que se trata de un fuente en lenguaje C# y nos mostrará cada palabra en un color que nos ayude a saber la misión de esa palabra:



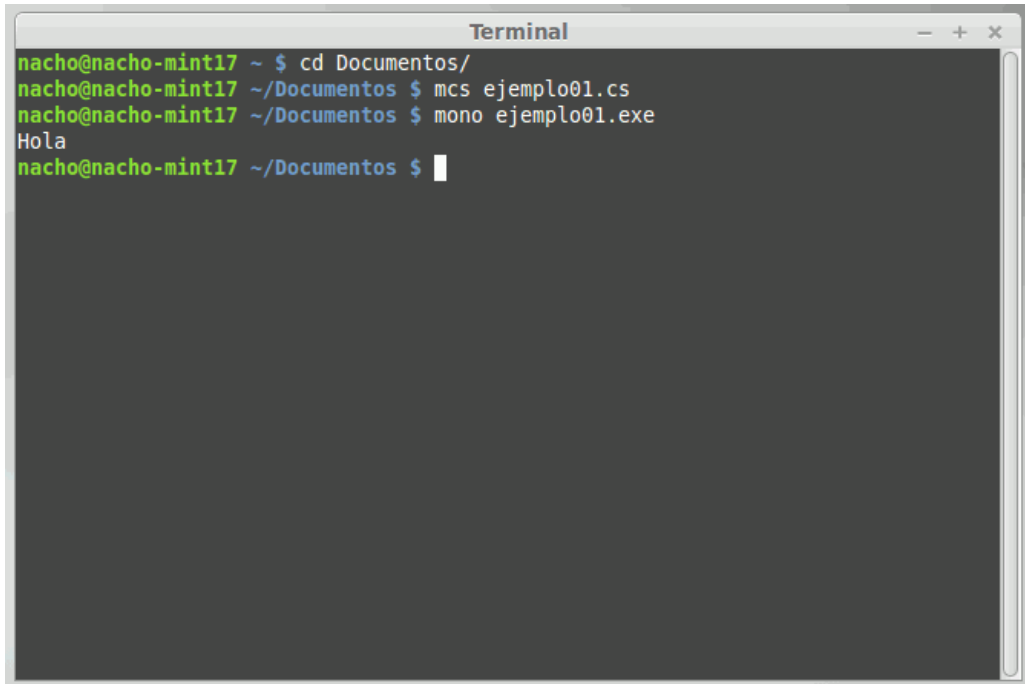
Para compilar y lanzar el programa usaremos un "terminal", que habitualmente estará accesible en la parte inferior de la pantalla:



En esa "pantalla negra" ya podemos teclear las órdenes necesarias para compilar y probar el programa:

- Si hemos guardado el fuente en la carpeta "Documentos", el primer paso será entrar a esa carpeta con la orden "cd Documentos".
- Después lanzaremos el compilador con la orden "mcs" seguida del nombre del fuente: "mcs ejemplo01.cs" (recuerda que en Linux debes respetar las mayúsculas y minúsculas tal y como las hayas escrito en el nombre del fichero).

- Si no aparece ningún mensaje de error, ya podemos lanzar el programa ejecutable, con la orden "mono" seguida del nombre del programa (terminado en ".exe"): "mono ejemplo01.exe", así:

A terminal window titled "Terminal" with standard window controls. The prompt is "nacho@nacho-mint17 ~". The user enters "cd Documentos/" and the prompt changes to "~/Documentos". Then the user enters "mcs ejemplo01.cs" and the prompt returns to "~/Documentos". Finally, the user enters "mono ejemplo01.exe" and the terminal outputs "Hola" on the next line. The prompt is now "~/Documentos \$" with a cursor.

```
nacho@nacho-mint17 ~ $ cd Documentos/  
nacho@nacho-mint17 ~/Documentos $ mcs ejemplo01.cs  
nacho@nacho-mint17 ~/Documentos $ mono ejemplo01.exe  
Hola  
nacho@nacho-mint17 ~/Documentos $
```

Si alguno de los pasos ha fallado, tendrás que comprobar si has dado los pasos anteriores de forma correcta y si tu fuente está bien tecleado.

Ejercicio propuesto (1.3.1.1): Desde Linux, crea y prueba un programa en C# que escriba en pantalla "Bienvenido a Linux".

1.3.2. Cómo probarlo con Mono en Windows

Si (utilices Linux o no) has leído las instrucciones para compilar usando Linux, tienes mucho trabajo adelantado. Instalar Mono para Windows no es mucho más complicado que para Linux, pero deberemos descargarlo desde su página oficial y responder a más preguntas durante la instalación. Además, tendremos (inicialmente) un editor de texto mucho más pobre que el de Linux.

Comenzaremos por descargar Mono desde su página oficial:

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



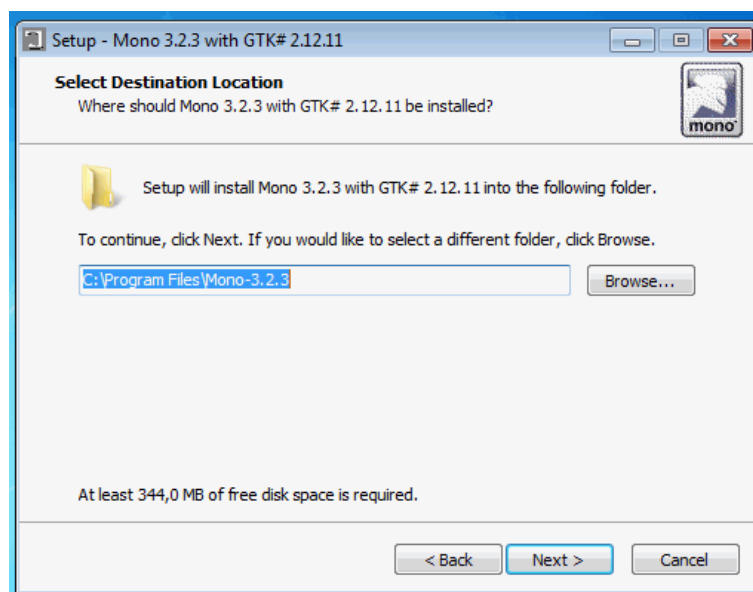
En la parte superior derecha aparece el enlace para descargar ("download"), que nos lleva a una nueva página en la que debemos elegir la plataforma para la que queremos nuestro Mono. Nosotros descargaremos la versión más reciente para Windows (la 3.4.0 en el momento de escribir este texto, aunque esa es la numeración para Mac y realmente en Windows corresponde a la 3.2.3).



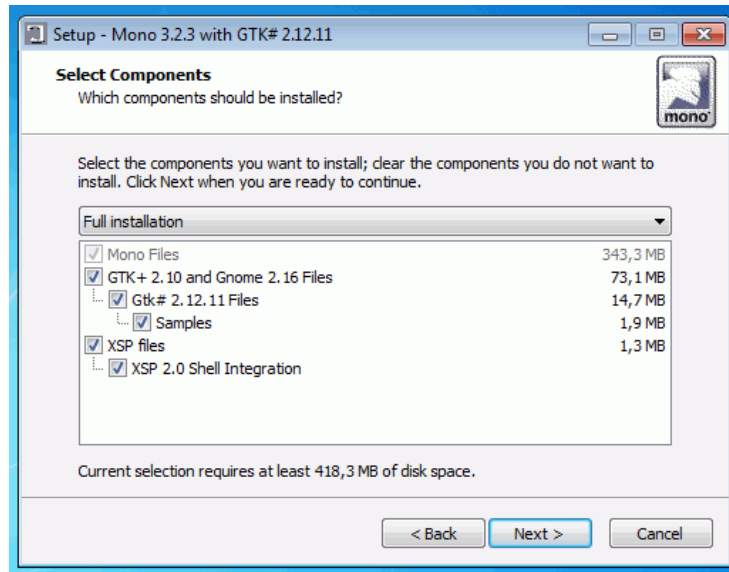
Se trata de un fichero de cerca de 100 Mb. Cuando termine la descarga, haremos doble clic en el fichero recibido, aceptaremos el aviso de seguridad que posiblemente nos mostrará Windows, y comenzará la instalación, en la que primero se nos muestra el mensaje de bienvenida:



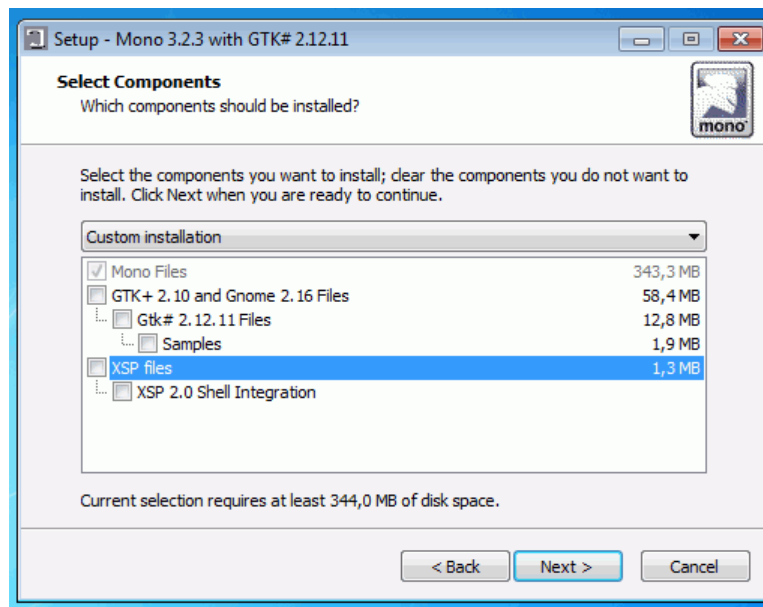
Deberemos aceptar el acuerdo de licencia y después se nos muestra una ventana de información y se nos pregunta en qué carpeta queremos instalar. Como es habitual, se nos propone que sea dentro de "Archivos de programa", pero podemos cambiarla por otra carpeta o incluso por otra unidad de disco:



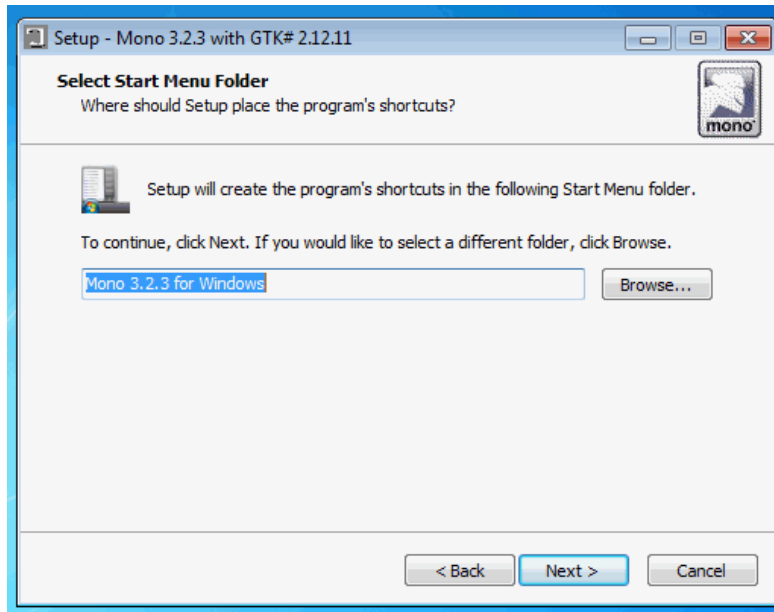
El siguiente paso es elegir qué componentes queremos instalar (Mono, Gtk#, XSP):



Yo no soy partidario de 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 investigar por su cuenta para 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:



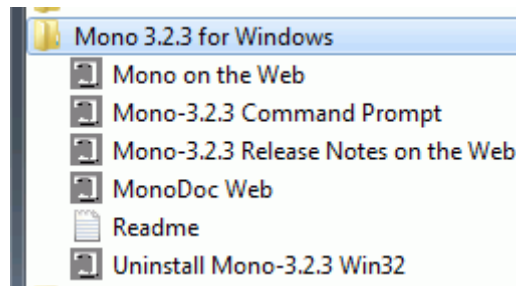
Después deberemos indicar en qué carpeta del menú de Inicio queremos que quede el acceso a Mono:



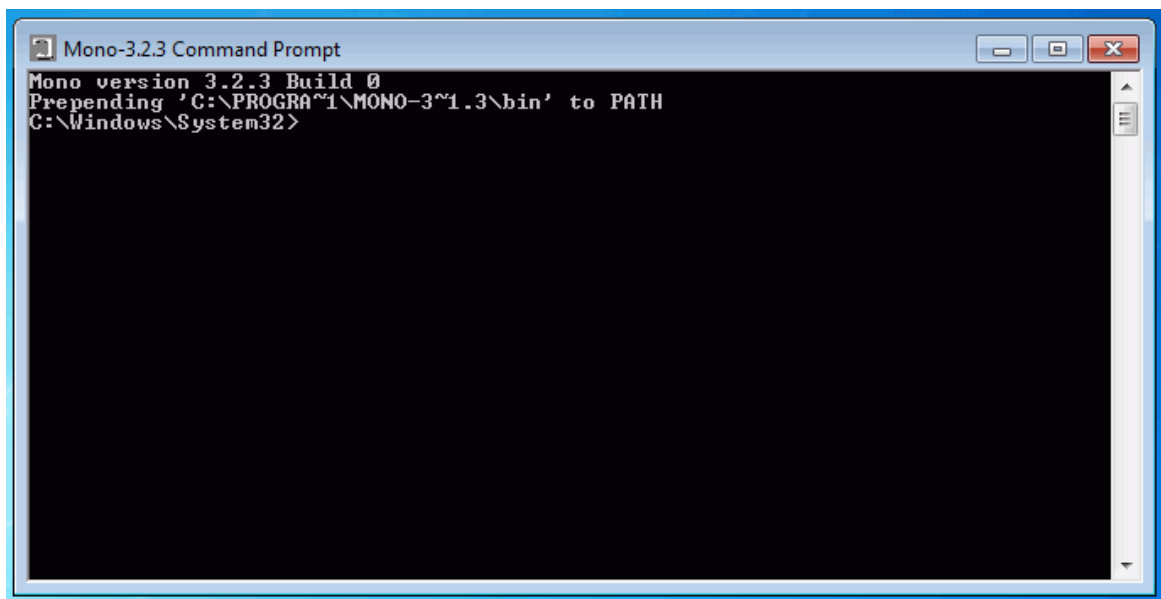
A continuación se nos muestra el resumen de lo que se va a instalar. Si confirmamos que todo nos parece correcto, comienza la copia de ficheros y al cabo de un instante tendremos el mensaje de confirmación de que la instalación se ha completado:



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



Quizá se nos lleve a una carpeta que esté dentro de "Documents and settings" o quizá (algo habitual en las últimas versiones) a alguna en la que no tengamos permiso para escribir, como "Windows\System32".

En ese caso, podemos crear una carpeta en nuestro escritorio, llamada (por ejemplo) "Programas", y comenzar por desplazarnos hasta ella, tecleando

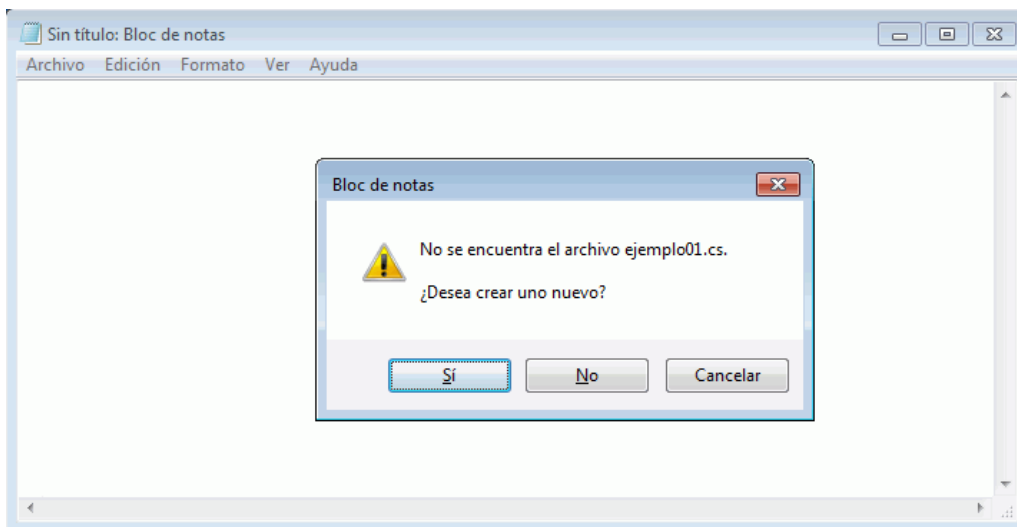
```
cd \users\yo\desktop\programas
```

(Esa sería la forma de hacerlo para un usuario llamado "Yo" en un sistema con Windows 7 o Windows 8; los detalles exactos dependerán del nombre del usuario y de la versión de Windows empleada).

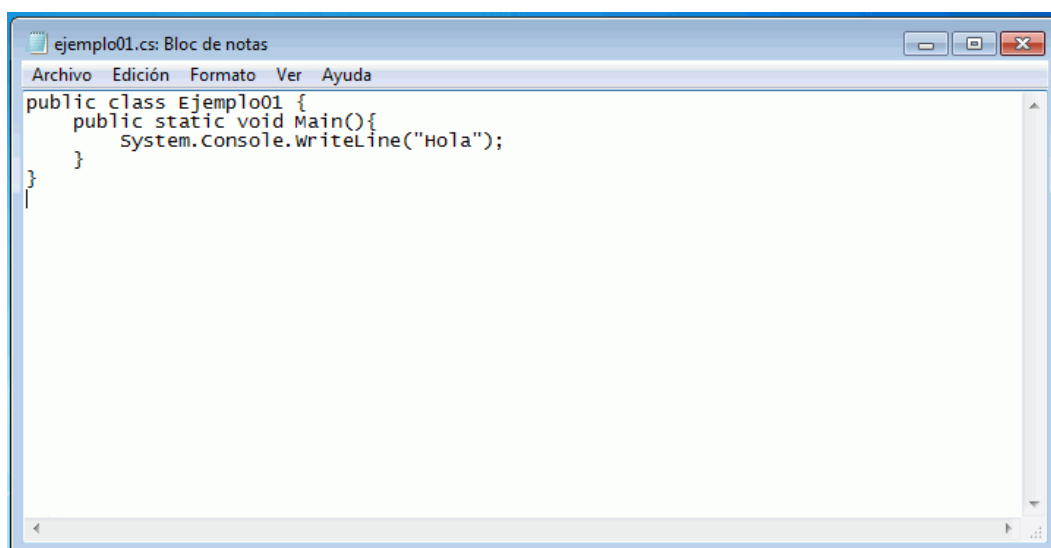
Para crear un programa, el primero paso será teclear el "fuente". Para ello podemos usar cualquier editor de texto. En este primer fuente, usaremos simplemente el "Bloc de notas" de Windows. Para ello, tecleamos:

```
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. Lo razonable es responder que sí:



Podemos empezar a teclear el ejemplo que habíamos visto anteriormente. En este caso no veremos ninguna palabra destacada en colores, ni siquiera después de guardar el programa, porque el "Bloc de notas" es mucho más limitado que los editores que incorporan los sistemas Linux. La alternativa será usar otro editor mejor, pero eso lo haremos más adelante.



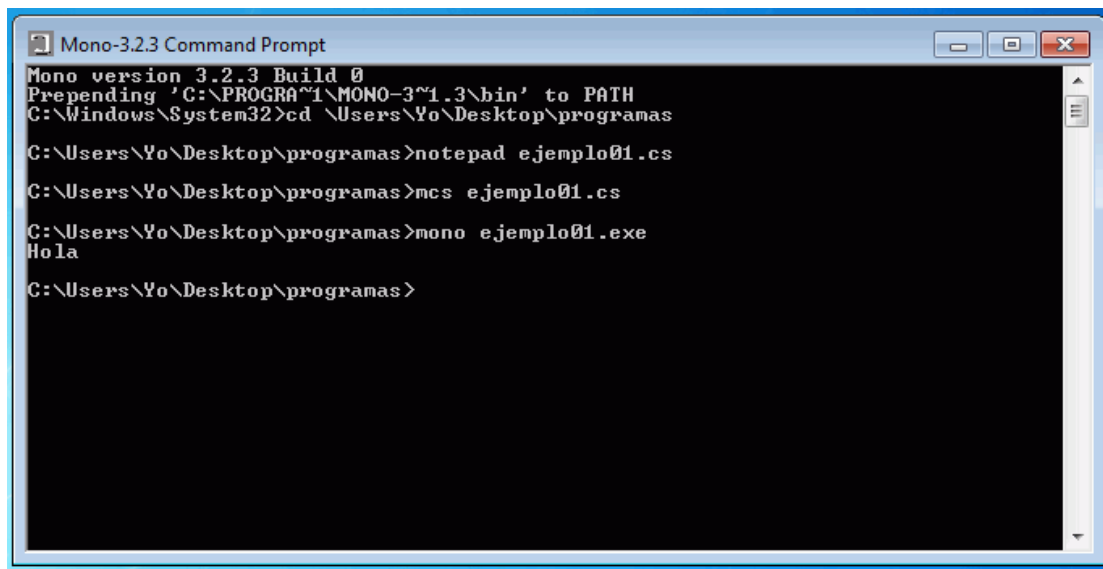
Guardamos los cambios, salimos del "Bloc de notas" y nos volvemos a encontrar en la pantalla negra del símbolo del sistema. Nuestro fuente ya está escrito y guardado. El siguiente paso es compilarlo. Para eso, tecleamos

```
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 ejemplo01.exe
```

y el mensaje "Hola" debería aparecer en pantalla.

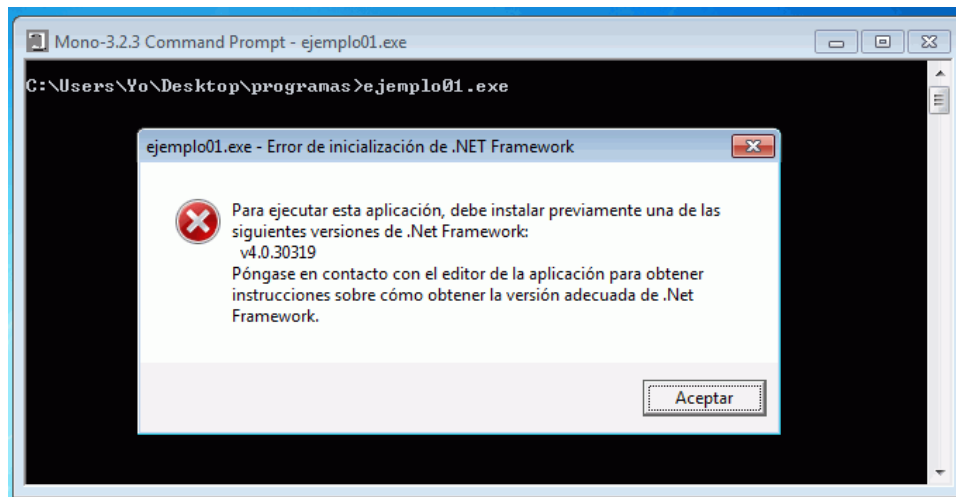


```
Mono-3.2.3 Command Prompt
Mono version 3.2.3 Build 0
Prepending 'C:\PROGRAM~1\MONO-3~1.3\bin' to PATH
C:\Windows\System32>cd \Users\Yo\Desktop\programas
C:\Users\Yo\Desktop\programas>notepad ejemplo01.cs
C:\Users\Yo\Desktop\programas>mcs ejemplo01.cs
C:\Users\Yo\Desktop\programas>mono ejemplo01.exe
Hola
C:\Users\Yo\Desktop\programas>
```

Si en nuestro ordenador está instalado el "Dot Net Framework" (algo que debería ser cierto en las últimas versiones de Windows, y que no ocurrirá en Linux ni Mac OSX), quizá no sea necesario decir que queremos que sea Mono quien lance nuestro programa, y podamos ejecutarlo directamente con su nombre:

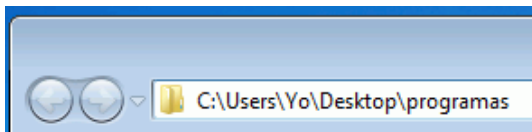
```
ejemplo01
```

Pero en ocasiones puede ocurrir que el ejecutable sea para una versión de la plataforma ".Net" distinta de la que tenemos instalada. En ese caso, tendremos que lanzarlo usando Mono, o bien deberemos descargar e instalar la correspondiente versión de ".Net" (es una descarga gratuita desde la página web de Microsoft):



Si el "Mono Command Prompt" se nos abre en Windows\System32 y queremos evitar tener que comenzar siempre con la orden "cd" para cambiar a nuestra carpeta de programas, tenemos una alternativa: cambiar la carpeta de arranque de Mono.

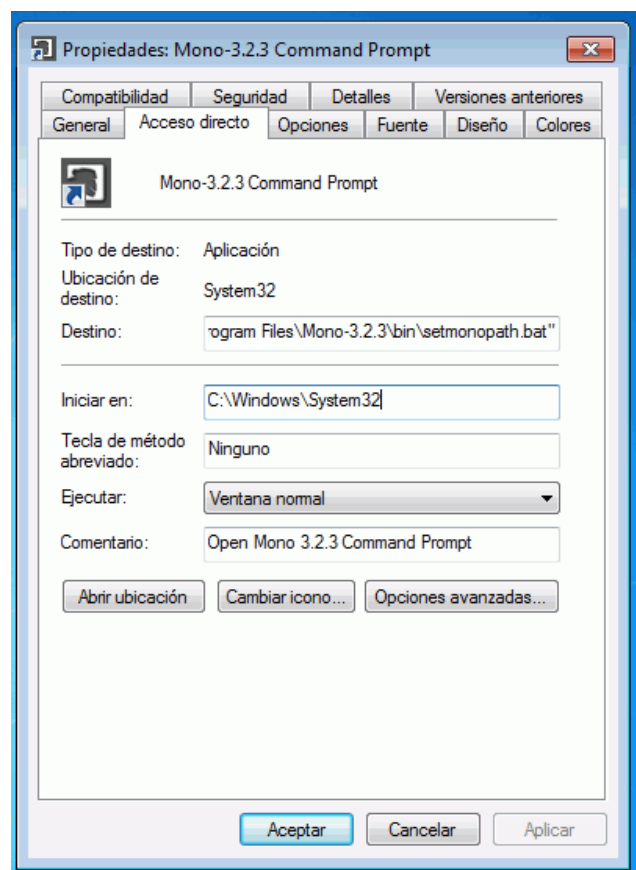
Es algo fácil en la mayoría de versiones de Windows. El primer paso será abrir la carpeta que hemos creado y hacer clic en su barra superior, de modo que se nos muestre la ruta completa de la carpeta.



Podemos "copiar" esa ruta (botón derecho del ratón o Ctrl+C) para memorizarla.

Ahora vamos al menú de Inicio de Windows, pulsamos el botón derecho sobre la opción "Mono-x.x.x Command Prompt" del menú de inicio y escogemos "Propiedades".

En la pestaña llamada "Acceso directo" veremos la opción "Iniciar en". Si cambiamos el contenido de esa casilla por el nombre de nuestra carpeta ("pegando" lo que antes habíamos copiado) conseguiremos que el



Command Prompt de Mono se abra en la carpeta que habíamos preparado para nuestros fuentes.

1.3.3. Otros editores más avanzados para Windows

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. **Geany** también es una alternativa muy interesante, disponible para muchos sistemas operativos, y que permite incluso compilar y lanzar los programas desde el propio editor, sin necesidad de salir al "intérprete de comandos" del sistema operativo. Un poco más adelante veremos cómo instalarlo y configurarlo, pero todavía no, para que durante los primeros temas tengas la obligación de repetir el proceso "teclear - compilar - ejecutar - probar" de forma un tanto "artesanal".

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 un apartado posterior hablaremos de **SharpDevelop** (para Windows) y de **MonoDevelop** (para Windows, Linux y Mac), así como del entorno "oficial" de desarrollo, llamado **Visual Studio**.

Hay un **posible problema** que se debe tener en cuenta: algunos de estos entornos de desarrollo muestran el resultado de nuestro programa y luego regresan al editor tan rápido que no da tiempo a ver los resultados. Una solución provisional puede ser añadir "System.Console.ReadLine()" al final del programa, de modo que se quede parado hasta que pulsemos Intro:

```
public class Ejemplo_01_03_03a
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
        System.Console.ReadLine();
    }
}
```

(**Nota:** no deberás entregar tus programas con ese "ReadLine" al final: es una ayuda para que compruebes que están funcionando correctamente desde ciertos entornos, pero no debería formar parte de un programa finalizado, porque no es parte de la lógica del programa).

1.4. Mostrar números enteros en pantalla

Cuando queremos escribir un texto "tal cual", como en el ejemplo anterior, lo encerramos entre comillas. Pero no siempre queremos 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 simple: no usar comillas en WriteLine. Entonces, C# intentará analizar el contenido para ver qué puede significar. Por ejemplo, para sumar 3 y 4 bastaría hacer:

```
public class Ejemplo_01_04a
{
    public static void Main()
    {
        System.Console.WriteLine(3+4);
    }
}
```

Ejercicios propuestos:

(1.4.1) Crea un programa que diga el resultado de sumar 118 y 56.

(1.4.2) Crea un programa que diga el resultado de sumar 12345 y 67890.

(Recomendación: no "copies y pegues" aunque dos ejercicios se parezcan. Volver a teclear cada nuevo ejercicio te ayudará a memorizar las estructuras básicas del lenguaje).

1.5. Operaciones aritméticas básicas

1.5.1. Operadores

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

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

Así, podemos calcular el resto de la división entre dos números de la siguiente forma:

```
public class Ejemplo_01_05_01a
{
    public static void Main()
    {
        System.Console.WriteLine("El resto de dividir 19 entre 5 es");
        System.Console.WriteLine(19 % 5);
    }
}
```

Ejercicios propuestos:

(1.5.1.1) Haz un programa que calcule el producto de los números 12 y 13.

(1.5.1.2) Un programa que calcule la diferencia (resta) entre 321 y 213.

(1.5.1.3) Un programa que calcule el resultado de dividir 301 entre 3.

(1.5.1.4) Un programa que calcule el resto de la división de 301 entre 3.

1.5.2. 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.

Así, el siguiente ejemplo da como resultado 23 (primero se multiplica $4*5$ y luego se le suma 3) en vez de 35 (no se suma $3+4$ antes de multiplicar, aunque aparezca a la izquierda, porque la prioridad de la suma es menor que la de la multiplicación).

```
public class Ejemplo_01_05_02a
{
    public static void Main()
    {
        System.Console.WriteLine("Ejemplo de precedencia de operadores");
        System.Console.WriteLine("3+4*5=");
        System.Console.WriteLine(3+4*5);
    }
}
```

Ejercicios propuestos: Calcular (a mano y después comprobar desde C#) el resultado de las siguientes operaciones:

- (1.5.2.1) Calcular el resultado de $-2 + 3 * 5$
- (1.5.2.2) Calcular el resultado de $(20+5) \% 6$
- (1.5.2.3) Calcular el resultado de $15 + -5*6 / 10$
- (1.5.2.4) Calcular el resultado de $2 + 10 / 5 * 2 - 7 \% 1$

1.5.3. Introducción a los problemas de desbordamiento

El espacio del que disponemos para almacenar los números es limitado. Veremos los límites exactos más adelante, pero de momento nos basta saber que 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 Ejemplo_01_05_03a
{
    public static void Main()
    {
        System.Console.WriteLine(10000000*10000000);
    }
}
```

1.6. Introducción a las variables: int

El primer ejemplo nos permitía escribir "Hola". El segundo llegaba un poco más allá y 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 reservar zonas de memoria a las que le demos un nombre y en las que guardemos los datos con los que vamos a trabajar y también los resultados temporales. A estas "zonas de memoria con nombre" les llamaremos **variables**.

Como primer ejemplo, vamos a ver lo que haríamos para sumar dos números enteros que fijásemos en el programa.

1.6.1. Definición de variables: números enteros

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

Ejercicio propuesto (1.6.1.1): Amplía el "Ejemplo 01.05.02a" para declarar tres variables, llamadas n1, n2, n3.

1.6.2. Asignación de valores

Podemos darle un valor a esa variable durante el programa haciendo

```
int primerNumero;
...
primerNumero = 234;
```

Hay que tener en cuenta que esto **no es una igualdad matemática**, sino una "asignación de valor": el elemento de la izquierda recibe el valor que indicamos a la derecha. Por eso **no se puede hacer `234 = primerNumero`**, y sí se puede cambiar el valor de una variable tantas veces como queramos

```
primerNumero = 234;
primerNumero = 237;
```

También podemos dar un valor inicial a las variables ("inicializarlas") antes de que empiece el programa, en el mismo momento en que las definimos:

```
int primerNumero = 234;
```

Si varias variables son del mismo tipo, podemos declararlas a la vez

```
int primerNumero, segundoNumero;
```

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;
```

Ejercicio propuesto (1.6.2.1): Amplía el ejercicio 1.6.1.1, para que las tres variables `n1`, `n2`, `n3` estén declaradas en la misma línea y tengan valores iniciales.

1.6.3. 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 (sin comillas, para que el compilador analice su valor de antes de escribir):

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

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

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

Si queremos mostrar de más de una variable, detallaremos en el texto dónde debe aparecer cada una de ellas, usando `{0}`, `{1}` y tantos números sucesivos como sea necesario, y tras el texto incluiremos los nombres de cada una de esas variables, separados por comas:

```
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 Ejemplo_01_06_03a
```

```

{
    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:

- (Aplazamos todavía los detalles de qué significan "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).

El resultado de este programa sería:

La suma de 234 y 567 es 801

Ejercicios propuestos:

(1.6.3.1) Crea un programa que calcule el producto de los números 121 y 132, usando variables.

(1.6.3.2) Crea un programa que calcule la suma de 285 y 1396, usando variables.

(1.6.3.3) Crea un programa que calcule el resto de dividir 3784 entre 16, usando variables.

(1.6.3.4) Amplía el ejercicio 1.6.2.1, para que se muestre el resultado de la operación $n1+n2*n3$.

1.7. Identificadores

Los nombres de variables (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 intermedios. También hay que recordar que las vocales acentuadas y la ñe son problemáticas, porque no son letras "estándar" en todos los idiomas, así que no se pueden utilizar como parte de un identificador en la mayoría de lenguajes de programación.

Por eso, no son nombres de variable válidos:

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

(Nota: algunos entornos de programación modernos sí permitirán variables que contengan ñe y vocales acentuadas, pero como no es lo habitual en todos los lenguajes de programación, durante este curso introductorio nosotros no consideraremos válido un nombre de variable como "año", aun sabiendo que si estamos programando en C# con Visual Studio, el sistema sí lo consideraría aceptable).

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).

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

```
PrimerNumero = 234;
primernumero = 234;
```

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


```
int primerNumero;
```

Ejercicios propuestos:

(1.7.1) Crea un programa que calcule el producto de los números 87 y 94, usando variables llamadas "numero1" y "numero2".

(1.7.2) Intenta crear una nueva versión del programa que calcula el producto de los números 87 y 94, usando esta vez variables llamadas "1numero" y "2numero".

(1.7.3) Intenta crear una nueva versión del programa que calcula el producto de los números 87 y 94, usando esta vez variables llamadas "numero 1" y "numero 2".

(1.7.4) Crea una nueva versión del programa que calcula el producto de los números 87 y 94, usando esta vez variables llamadas "número1" y "número2".

1.8. Comentarios

Podemos escribir comentarios, que el compilador ignorará, pero que pueden ser útiles para nosotros mismos, haciendo que sea más fácil recordar el cometido un fragmento del programa más adelante, cuando tengamos que ampliarlo o corregirlo.

Existen dos formas de indicar comentarios. En su forma más general, los escribiremos entre `/*` y `*/`:

```
int suma; /* 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 podría ser:

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

public class Ejemplo_01_08a
{
    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 de una línea" o "comentarios al estilo de C++" (a diferencia de los "comentarios de múltiples líneas" o "comentarios al estilo de C" que ya hemos visto):

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

De modo que el programa anterior se podría reescribir usando comentarios de una línea:

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

public class Ejemplo_01_08b
{
    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);
    }
}
```

En este texto, a partir de ahora los fuentes comenzarán con un comentario que resuma su cometido, y en ocasiones incluirán también comentarios intermedios.

Ejercicios propuestos:

(1.8.1) Crea un programa que convierta una cantidad prefijada de metros (por ejemplo, 3000) a millas. La equivalencia es 1 milla = 1609 metros. Usa comentarios donde te parezca adecuado.

1.9. Datos por el usuario: ReadLine

Hasta ahora hemos utilizado datos prefijados, pero eso es poco frecuente en el mundo real. Es mucho más habitual que los datos los introduzca el usuario, o que se lean desde un fichero, o desde una base de datos, o se reciban de Internet o cualquier otra red. El primer caso que veremos será el de interaccionar directamente con el usuario.

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 un poco más adelante, 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:

```
// Ejemplo en C#: sumar dos números introducidos por el usuario
public class Ejemplo_01_09a
{
    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);
    }
}
```

Ejercicios propuestos:

(1.9.1) Crea un programa que calcule el producto de dos números introducidos por el usuario.

(1.9.2) Crea un programa que calcule la división de dos números introducidos por el usuario, así como el resto de esa división.

(1.9.3) Suma tres números tecleados por usuario.

(1.9.4) Pide al usuario una cantidad de "millas náuticas" y muestra la equivalencia en metros, usando: 1 milla náutica = 1852 metros.

1.10. *using System*

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":

```
// Ejemplo en C#: "using System" en vez de "System.Console"
using System;

public class Ejemplo_01_10a
{
    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);
    }
}
```

Si además declaramos varias variables a la vez, como vimos en el apartado 1.5.2, el programa podría ser aún más compacto:

```
// Ejemplo en C#: "using System" y declaraciones múltiples de variables
using System;

public class Ejemplo_01_10b
{
    public static void Main()
    {
        int primerNumero, segundoNumero, 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:

(1.10.1) Crea una nueva versión del programa que calcula el producto de dos números introducidos por el usuario (1.9.1), empleando "using System". El programa deberá contener un comentario al principio, que recuerde cual es su objetivo.

(1.10.2) Crea una nueva versión del programa que calcula la división de dos números introducidos por el usuario, así como el resto de esa división (1.9.2), empleando "using System". Deberás incluir un comentario con tu nombre y la fecha en que has realizado el programa.

1.11. Escribir sin avanzar de línea

En el apartado 1.6.3 vimos cómo usar {0} para escribir en una misma línea datos calculados y textos prefijados. Pero hay otra alternativa, que además nos permite también escribir un texto y pedir un dato a continuación, en la misma línea de pantalla: emplear "Write" en vez de "WriteLine", así:

```

// Ejemplo en C#: escribir sin avanzar de línea
using System;

public class Ejemplo_01_11a
{
    public static void Main()
    {
        int primerNumero, segundoNumero, suma;

        Console.Write("Introduce el primer número: ");
        primerNumero = Convert.ToInt32(Console.ReadLine());
        Console.Write("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);
    }
}

```

Incluso el último "WriteLine" de varios datos se podría convertir en varios Write (aunque generalmente eso hará el programa más largo y no necesariamente más legible), así

```
// Ejemplo en C#: escribir sin avanzar de línea (2)
using System;

public class Ejemplo_01_11b
{
    public static void Main()
    {
        int primerNumero, segundoNumero, suma;

        Console.Write("Introduce el primer número: ");
        primerNumero = Convert.ToInt32(Console.ReadLine());
        Console.Write("Introduce el segundo número: ");
        segundoNumero = Convert.ToInt32(Console.ReadLine());
        suma = primerNumero + segundoNumero;
        Console.Write("La suma de ");
        Console.Write(primerNumero);
        Console.Write(" y ");
        Console.Write(segundoNumero);
        Console.Write(" es ");
        Console.WriteLine(suma);
    }
}
```

Ejercicios propuestos:

- **(1.11.1)** 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 . Ambos resultados se deben mostrar en la misma línea.
- **(1.11.2)** Pedir al usuario un número y mostrar su tabla de multiplicar, usando {0},{1} y {2}. Por ejemplo, si el número es el 3, debería escribirse algo como


```
3 x 0 = 0
3 x 1 = 3
3 x 2 = 6
...
3 x 10 = 30
```
- **(1.11.3)** Crear una variante del programa anterior, que pide al usuario un número y muestra su tabla de multiplicar. Esta vez no deberás utilizar {0}, {1}, {2}, sino "Write".
- **(1.11.4)** Crea un programa que convierta de grados Celsius (centígrados) a Kelvin y a Fahrenheit: pedirá al usuario la cantidad de grados centígrados y usará las siguiente tablas de conversión: $kelvin = celsius + 273$; $fahrenheit = celsius \times 18 / 10 + 32$. Emplea "Write" en vez de "{0}" cuando debas mostrar varios datos en la misma línea.

2. Estructuras de control

Casi cualquier problema del mundo real que debamos resolver o tarea que deseemos automatizar supondrá tomar decisiones: dar una serie de pasos en función de si se cumplen ciertas condiciones o no. En muchas ocasiones, además esos pasos deberán ser repetitivos. Vamos a ver cómo podemos comprobar si se cumplen condiciones y también cómo hacer que un bloque de un programa se repita.

2.1. Estructuras alternativas

2.1.1. if

La primera construcción que emplearemos para comprobar si se cumple una condición será "**si ... entonces ...**". Su formato es

```
if (condición) sentencia;
```

Es decir, debe empezar con la palabra "if", la condición se debe indicar entre paréntesis y a continuación se detallará la orden que hay que realizar en caso de cumplirse esa condición, terminando con un punto y coma.

Vamos a verlo con un ejemplo:

```
// Ejemplo_02_01_01a.cs
// Condiciones con if
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_01a
{
    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:

(2.1.1.1) Crea 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) ...`).

(2.1.1.2) Crea un programa que pida al usuario dos números enteros y diga cuál es el mayor de ellos.

(2.1.1.3) Crea 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_02_01_02a.cs
// Condiciones con if (2): Sentencias compuestas
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_02_01_02a
{
    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.");
    } // Aquí acaba el "if"
    } // Aquí acaba "Main"
} // Aquí acaba "Ejemplo06"

```

En este caso, si el número es positivo, se hacen dos cosas: escribir un texto y luego... ¡escribir otro! (no es gran cosa; 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 aún más a la derecha. Este "**sangrado**" del texto se suele llamar "**escritura indentada**". Un tamaño habitual para el sangrado es de 4 espacios, aunque en este texto en algunas ocasiones usaremos sólo dos espacios, para que fuentes más complejos quepan entre los márgenes del papel.

Ejercicios propuestos:

(2.1.2.1) Crea un programa que pida al usuario un número entero. Si es múltiplo de 10, informará al usuario y pedirá un segundo número, para decir a continuación si este segundo número también es múltiplo de 10.

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_02_01_03a.cs
// Condiciones con if (3): "distinto de"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_03a
{
    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:

(2.1.3.1) Crea 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.

(2.1.3.2) Crea un programa que pida al usuario dos números enteros. 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_02_01_04a.cs
// Condiciones con if (4): caso contrario ("else")
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_04a
{
    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_02_01_04b.cs
// Condiciones con if (5): caso contrario, sin "else"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_04b
{
    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 02_01_04a) 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 02_01_04b), 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_02_01_04c.cs
// Condiciones con if (6): condiciones encadenadas
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_04c
{

```

```

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:

(2.1.4.1) Mejora la solución al ejercicio 2.1.3.1, usando "else".

(2.1.4.2) Mejora la solución al ejercicio 2.1.3.2, usando "else".

2.1.5. Operadores lógicos: &&, ||, !

Las condiciones se puede **encadenar** con "y", "o", "no". Por ejemplo, una partida de un juego puede acabar si nos quedamos sin vidas **o** si superamos el último nivel. Y podemos avanzar al nivel siguiente si hemos llegado hasta la puerta **y** hemos encontrado la llave. O deberemos volver a pedir una contraseña si **no** es correcta **y no** hemos agotado los intentos.

Esos operadores se indican de la siguiente forma

<i>Operador</i>	<i>Significado</i>
&&	Y
	O
!	No

De modo que, ya con la sintaxis de C#, podremos escribir cosas como

```

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

```

Así, un programa que dijera si dos números introducidos por el usuario son cero, podría ser:

```

// Ejemplo_02_01_05a.cs
// Condiciones con if enlazadas con &&
// Introducción a C#, por Nacho Cabanes

```

```

using System;

```

```

public class Ejemplo_02_01_05a
{
    public static void Main()
    {
        int n1, n2;

        Console.Write("Introduce un número: ");
        n1 = Convert.ToInt32(Console.ReadLine());
        Console.Write("Introduce otro número: ");
        n2 = Convert.ToInt32(Console.ReadLine());

        if ((n1 > 0) && (n2 > 0))
            Console.WriteLine("Ambos números son positivos.");
        else
            Console.WriteLine("Al menos uno no es positivo.");
    }
}

```

Una curiosidad: en C# (y en algún otro lenguaje de programación), la evaluación de dos condiciones que estén enlazadas con "Y" se hace "en **cortocircuito**": si la primera de las condiciones no se cumple, ni siquiera se llega a comprobar la segunda, porque se sabe de antemano que la condición formada por ambas no podrá ser cierta. Eso supone que en el primer ejemplo anterior, `if ((opcion==1) && (usuario==2))`, si "opcion" no vale 1, el compilador no se molesta en ver cuál es el valor de "usuario", porque, sea el que sea, no podrá hacer que sea "verdadera" toda la expresión. Lo mismo ocurriría si hay dos condiciones enlazadas con "o", y la primera de ellas es "verdadera": no será necesario comprobar la segunda, porque ya se sabe que la expresión global será "verdadera".

Como la mejor forma de entender este tipo de expresiones es practicándolas, vamos a ver unos cuantos ejercicios propuestos...

Ejercicios propuestos:

- (2.1.5.1) Crea un programa que pida al usuario un número entero y responda si es múltiplo de 2 o de 3.
- (2.1.5.2) Crea un programa que pida al usuario un número entero y responda si es múltiplo de 2 y de 3 simultáneamente.
- (2.1.5.3) Crea un programa que pida al usuario un número entero y responda si es múltiplo de 2 pero no de 3.
- (2.1.5.4) Crea un programa que pida al usuario un número entero y responda si no es múltiplo de 2 ni de 3.
- (2.1.5.5) Crea un programa que pida al usuario dos números enteros y diga si ambos son pares.
- (2.1.5.6) Crea un programa que pida al usuario dos números enteros y diga si (al menos) uno es par.

(2.1.5.7) Crea un programa que pida al usuario dos números enteros y diga si uno y sólo uno es par.

(2.1.5.8) Crea 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.

(2.1.5.9) Crea un programa que pida al usuario tres números y muestre cuál es el mayor de los tres.

(2.1.5.10) Crea un programa que pida al usuario dos números enteros y diga si son iguales o, en caso contrario, cuál es el mayor de ellos.

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

Cuidado con el comparador 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 al menos nos avisarían de que quizá estemos asignando un valor sin pretenderlo, pero no es un "error" que invalide la compilación, 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 sea "verdadero" o "falso" (lo que pronto llamaremos 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_02_01_06a.cs
// Condiciones con if: comparación incorrecta
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_06a
{
    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.");
    }
}

```

Nota: en lenguajes como C y C++, en los que sí existe este riesgo de asignar un valor en vez de comparar, se suele recomendar plantear la comparación al revés, colocando el número en el lado izquierdo, de modo que si olvidamos el doble signo de "=", obtendríamos una asignación no válida y el programa no compilaría:

```
if (0 == numero) ...
```

Ejercicios propuestos:

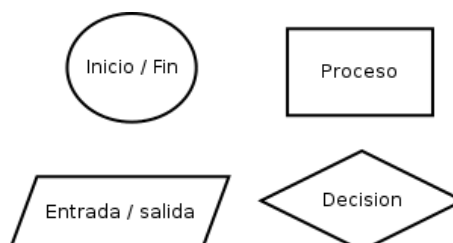
(2.1.6.1) Crea una variante del ejemplo 02_01_06a, en la que la comparación de igualdad sea correcta y en la que las variables aparezcan en el lado derecho de la comparación y los números en el lado izquierdo.

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.

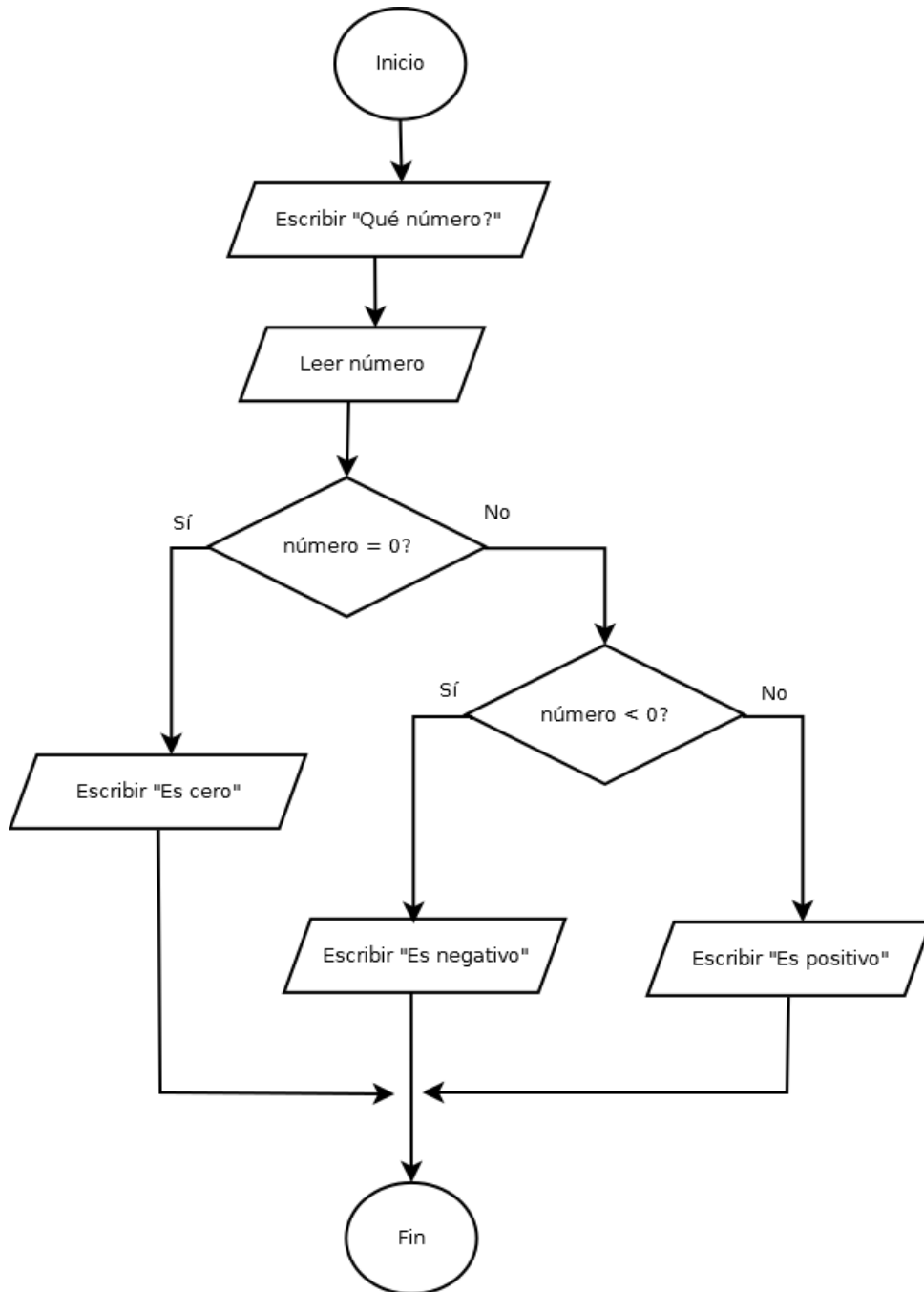


Es decir:

- 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 lados superior e inferior horizontales, pero no tenga verticales los otros dos.
- Las decisiones se indican dentro de un rombo, desde el que saldrán dos flechas. Cada una de ellas corresponderá a la secuencia de pasos a dar si se cumple una de las dos opciones posibles.

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 "sí" y si no se cumple ("else") ejecutará lo que aparezca en su salida "no".

Eso sí, hay que tener en cuenta que ésta es una **notación anticuada**, y que no permite representar de forma fiable las estructuras repetitivas que veremos dentro de poco, por lo que su uso actual es muy limitado.

Ejercicios propuestos:

(2.1.7.1) Crea el diagrama de flujo para el programa que pide dos números al usuario y dice cuál es el mayor de los dos.

(2.1.7.2) Crea 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.

(2.1.7.3) Crea 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#, al igual que en la mayoría de lenguajes que derivan de C, hay otra forma de asignar un valor según se cumpla una condición o no, más compacta pero también más difícil de leer. Es el "**operador condicional**" **?** : (también conocido como "operador ternario"), 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;
```

Al igual que en este ejemplo, podremos usar el operador condicional cuando queramos optar entre dos valores posibles para una variable, dependiendo de si se cumple o no una condición.

Aplicado a un programa sencillo, podría ser

```
// Ejemplo_02_01_08a.cs
// El operador condicional
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_02_01_08a
{
    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);
}
}

```

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

```

// Ejemplo_02_01_08b.cs
// El operador condicional (2)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_08b
{
    public static void Main()
    {
        int a, b, operacion, resultado;

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

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

        Console.Write("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}.", resultado);
    }
}

```

Ejercicios propuestos:

- (2.1.8.1)** Crea 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.
- (2.1.8.2)** Usa el operador condicional para calcular el menor de dos números.

2.1.9. switch

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

```
switch (expresión)
{
    case valor1: sentencia1;
        break;
    case valor2: sentencia2;
        sentencia2b;
        break;
    case valor3:
        goto case valor1;
    ...
    case valorN: sentenciaN;
        break;
    default:
        otraSentencia;
        break;
}
```

Es decir:

- Tras la palabra "**switch**" se escribe 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 la expresión tiene un cierto 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 a ver 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_02_01_09a.cs
// La orden "switch" (1)
// Introducción a C#, por Nacho Cabanes
```

```
using System;

public class Ejemplo_02_01_09a
{
    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 (para evitar errores provocados por una "break" olvidado) con la **única excepción** de que un caso no haga absolutamente nada excepto dejar pasar el control al siguiente caso, y en ese caso se puede dejar totalmente vacío:

```

// Ejemplo_02_01_09b.cs
// La orden "switch" (variante sin break)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_09b
{
    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_02_01_09c.cs
// La orden "switch" con cadenas de texto
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_01_09c
{
    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:

(2.1.9.1) Crea un programa que pida un número del 1 al 5 al usuario, y escriba el nombre de ese número, usando "switch" (por ejemplo, si introduce "1", el programa escribirá "uno").

(2.1.9.2) Crea un programa que lea una letra tecleada por el usuario y diga si se trata de un signo de puntuación (., ; :), una cifra numérica (del 0 al 9) o algún otro carácter, usando "switch" (pista: habrá que usar un dato de tipo "char").

(2.1.9.3) Crea un programa que lea una letra tecleada por el usuario y diga si se trata de una vocal, una cifra numérica o una consonante, usando "switch".

(2.1.9.4) Repite el ejercicio 2.1.9.1, empleando "if" en lugar de "switch".

(2.1.9.5) Repite el ejercicio 2.1.9.2, empleando "if" en lugar de "switch" (pista: como las cifras numéricas del 0 al 9 están ordenadas, no hace falta comprobar los 10 valores, sino que se puede hacer con "if ((simbolo >= '0') && (simbolo <='9'))").

(2.1.9.6) Repite el ejercicio 2.1.9.3, 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

2.2.1.1. Estructura básica de un bucle "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 del bloque repetitivo.

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 llaves: { y }.

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

```
// Ejemplo_02_02_01_01a.cs
// La orden "while": mientras...
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_01_01a
{
    public static void Main()
    {
        int numero;

        Console.Write("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:

(2.2.1.1.1) Crea un programa que pida al usuario su contraseña (numérica). Deberá terminar cuando introduzca como contraseña el número 1111, pero volvérsela a pedir tantas veces como sea necesario.

(2.2.1.1.2) Crea un "calculador de cuadrados": pedirá al usuario un número y mostrará su cuadrado. Se repetirá mientras el número introducido no sea cero (usa "while" para conseguirlo).

(2.2.1.1.3) Crea un programa que pida de forma repetitiva pares de números al usuario. Tras introducir cada par de números, responderá si el primero es múltiplo del segundo.

(2.2.1.1.4) Crea una versión mejorada del programa anterior, que, tras introducir cada par de números, responderá si el primero es múltiplo del segundo, o el segundo es múltiplo del primero, o ninguno de ellos es múltiplo del otro.

2.2.1.2. Contadores usando un bucle "while"

Ahora que sabemos "repetir" cosas, podemos utilizarlo también para **contar**. Por ejemplo, si queremos contar del 1 al 5, usaríamos una variable que empezase en 1, que aumentaría una unidad en cada repetición y se repetiría hasta llegar al valor 5, así:

```
// Ejemplo_02_02_01_02a.cs
// Contar con "while"
// Introducción a C#, por Nacho Cabanes
```

```
using System;

public class Ejemplo_02_02_01_02a
{
    public static void Main()
    {
        int n = 1;

        while (n < 6)
        {
            Console.WriteLine(n);
            n = n + 1;
        }
    }
}
```

Ejercicios propuestos:

(2.2.1.2.1) Crea un programa que escriba en pantalla los números del 1 al 10, usando "while".

(2.2.1.2.2) Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "while".

(2.2.1.2.3) Crea un programa calcule cuantas cifras tiene un número entero positivo (pista: se puede hacer dividiendo varias veces entre 10).

(2.2.1.2.4) Crea 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.

2.2.2. do ... while

Este es el otro formato que puede tener la orden "while": en este caso, la condición se comprueba **al final**, de modo que siempre se dará al menos una pasada por la zona repetitiva (se podría traducir como "repetir...mientras"). El punto en que comienza la parte repetitiva 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_02_02_02a.cs
// La orden "do..while" (repetir..mientras)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_02a
{
    public static void Main()
    {
        int valida = 711;
        int clave;

        do
        {
            Console.WriteLine("Introduzca su clave numérica: ");
            clave = Convert.ToInt32(Console.ReadLine());

            if (clave != valida)
                Console.WriteLine("No válida!");
        } while (clave != valida);

        Console.WriteLine("Aceptada.");
    }
}
```

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.

Como veremos con detalle un poco más adelante, si preferimos que la clave sea un texto en vez de un número, los cambios al programa son mínimos, basta con usar "string" e indicar su valor entre comillas dobles:

```
// Ejemplo_02_02_02b.cs
// La orden "do..while" (2)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_02b
{
    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!");
}
while (clave != valida);

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

```

Ejercicios propuestos:

(2.2.2.1) Crear un programa que pida números positivos al usuario, y vaya calculando y mostrando la suma de todos ellos (terminará cuando se teclea un número negativo o cero).

(2.2.2.2) Crea un programa que escriba en pantalla los números del 1 al 10, usando "do..while".

(2.2.2.3) Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "do..while".

(2.2.2.4) Crea un programa que pida al usuario su identificador y su contraseña (ambos numéricos), y no le permita seguir hasta que introduzca como identificador "1234" y como contraseña "1111".

(2.2.2.5) Crea un programa que pida al usuario su identificador 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, de modo que la forma habitual de crear el contador anterior sería

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

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_02_02_03a.cs
// Uso básico de "for"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_03a
{
    public static void Main()
    {
        int contador;

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

Ejercicios propuestos:

(2.2.3.1) Crea un programa que muestre los números del 10 al 20, ambos incluidos.

(2.2.3.2) Crea 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).

(2.2.3.1.3) Crea un programa que muestre los números del 100 al 200 (ambos incluidos) que sean divisibles entre 7 y a la vez entre 3.

(2.2.3.4) Crea un programa que muestre la tabla de multiplicar del 9.

(2.2.3.5) Crea un programa que muestre los primeros ocho números pares: 2 4 6 8 10 12 14 16 (pista: en cada pasada habrá que aumentar de 2 en 2, o bien mostrar el doble del valor que hace de contador).

(2.2.3.6) Crea 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--`).

2.2.4. Bucles sin fin

Realmente, en un "for", 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 nunca ("bucle sin fin") sería la siguiente orden:

```
for ( ; ; )
```

También se puede crear un bucle sin fin usando "while" y usando "do..while", si se indica una condición que siempre vaya a ser cierta, como ésta:

```
while (1 == 1)
```

Ejercicios propuestos:

(2.2.4.1) Crea un programa que contenga un bucle sin fin que escriba "Hola " en pantalla, sin avanzar de línea.

(2.2.4.2) Crea un programa que contenga un bucle sin fin que muestre los números enteros positivos a partir del uno.

2.2.5. Bucles anidados

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

```
// Ejemplo_02_02_05a.cs
// "for" anidados
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_02_02_05a
{
    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);
    }
}

```

(Es decir: tenemos varias tablas, del 1 al 5, y para cada tabla queremos ver el resultado que se obtiene al multiplicar por los números del 1 al 10).

Ejercicios propuestos:

(2.2.5.1) Crea un programa escriba 4 veces los números del 1 al 5, en una misma línea, usando "for": 12345123451234512345.

(2.2.5.2) Crea un programa escriba 4 veces los números del 1 al 5, en una misma línea, usando "while": 12345123451234512345.

(2.2.5.3) Crea un programa que, para los números entre el 10 y el 20 (ambos incluidos) diga si son divisibles entre 5, si son divisibles entre 6 y si son divisibles entre 7.

2.2.6. Repetir sentencias compuestas

En los últimos 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_02_02_06a.cs
// "for" anidados (2)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_06a
{
    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();
        }
    }
}

```

```
}
```

Ejercicios propuestos:

(2.2.6.1) Crea un programa que escriba 4 líneas de texto, cada una de las cuales estará formada por los números del 1 al 5.

(2.2.6.2) Crea un programa que pida al usuario el ancho (por ejemplo, 4) y el alto (por ejemplo, 3) y escriba un rectángulo formado por esa cantidad de asteriscos:

```
*****
*****
*****
```

2.2.7. Contar con letras

Para "contar" no necesariamente hay que usar números. Por ejemplo, podemos usar letras, si el contador lo declaramos como "char" y los valores inicial y final se detallan entre comillas simples, así:

```
// Ejemplo_02_02_07a.cs
// "for" que usa "char"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_07a
{
    public static void Main()
    {
        char letra;

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

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

Como ya hemos comentado, 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. Así, podríamos escribir las letras de la "z" a la "a" de la siguiente manera:

```
// Ejemplo_02_02_07b.cs
// "for" que descuenta
// Introducción a C#, por Nacho Cabanes

using System;
```

```
public class Ejemplo_02_02_07b
{
    public static void Main()
    {
        char letra;

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

Ejercicios propuestos:

(2.2.7.1) Crea un programa que muestre las letras de la Z (mayúscula) a la A (mayúscula, descendiendo).

(2.2.7.2) Crea un programa que muestre 5 veces las letras de la L (mayúscula) a la N (mayúscula), en la misma línea.

2.2.8. Declarar variables en un "for"

Se puede incluso declarar una nueva variable en el interior de "for", y esa variable dejará de estar definida cuando el "for" acabe. Es una forma recomendable de trabajar, porque ayuda a evitar un fallo frecuente: reutilizar variables pero olvidar volver a darles un valor inicial:

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

Por ejemplo, el siguiente fuente compila correctamente y puede parecer mostrar dos veces la tabla de multiplicar del 3, pero el "while" no muestra nada, porque no hemos vuelto a inicializar la variable "n", así que ya está por encima del valor 10 y ya no es un valor aceptable para entrar al bloque "while":

```
// Ejemplo_02_02_08a.cs
// Reutilizacion incorrecta de la variable de un "for"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_08a
{
    public static void Main()
    {
        int n = 1;
        // Vamos a mostrar la tabla de multiplicar del 3 con "for"
        for (n=1; n<=10; n++)
            Console.WriteLine("{0} x 3 = {1}", n, n*3);

        // Y ahora con "while"... pero no funcionará correctamente
        while (n<=10)
        {
            Console.WriteLine("{0} x 3 = {1}", n, n*3);
            n++;
        }
    }
}
```



```
    }
  }
}
```

Si declaramos la variable dentro del "for", la zona de "while" no compilaría, lo que hace que el error de diseño sea evidente:

```
// Ejemplo_02_02_08b.cs
// Intento de reutilizacion incorrecta de la variable
//   de un "for": no compila
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_08b
{
    public static void Main()
    {
        // Vamos a mostrar la tabla de multiplicar del 3 con "for"
        for (int n=1; n<=10; n++)
            Console.WriteLine("{0} x 3 = {1}", n, n*3);

        // Y ahora con "while"... pero no compila
        while (n<=10)
        {
            Console.WriteLine("{0} x 3 = {1}", n, n*3);
            n++;
        }
    }
}
```

Esta idea sea puede aplicar a cualquier fuente que contenga un "for". Por ejemplo, el fuente 2.2.6a, que mostraba varias tablas de multiplicar, se podría reescribir de forma más segura así:

```
// Ejemplo_02_02_08c.cs
// "for" anidados, variables en "for"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_08c
{
    public static void Main()
    {
        for (int tabla=1; tabla<=5; tabla++)
        {
            for (int numero=1; numero<=10; numero++)
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                    tabla*numero);

            Console.WriteLine();
        }
    }
}
```

Ejercicios propuestos:

(2.2.8.1) Crea un programa que escriba 6 líneas de texto, cada una de las cuales estará formada por los números del 1 al 7. Debes usar dos variables llamadas "línea" y "numero", y ambas deben estar declaradas en el "for".

(2.2.8.2) Crea un programa que pida al usuario el ancho (por ejemplo, 4) y el alto (por ejemplo, 3) y escriba un rectángulo formado por esa cantidad de asteriscos, como en el ejercicio 2.2.6.2. Deberás usar las variables "ancho" y "alto" para los datos que pidas al usuario, y las variables "filaActual" y "columnaActual" (declaradas en el "for") para el bloque repetitivo.

2.2.9. Las llaves son recomendables

Sabemos que las "llaves" no son necesarias cuando una orden "for" va a repetir una única sentencia, sino cuando se repite un bloque de dos o más sentencias, y que lo mismo ocurre con "while", "do-while" e "if". Pero un error frecuente es repetir inicialmente una única orden, añadir después una segunda orden repetitiva y olvidar las llaves. llaves. Por eso, una alternativa recomendable es incluir siempre las llaves, aunque esperemos repetir sólo una una orden.

Por ejemplo, el siguiente fuente puede parecer correcto, pero si lo miramos con detenimiento, veremos que la orden "Console.WriteLine" del final, aunque esté tabulada más a la derecha, no forma parte de ningún "for", de modo que no se repite, y no se dejará ningún espacio en blanco entre una tabla de multiplicar y la siguiente, sino que sólo se escribirá una línea en blanco al final, justo antes de terminar el programa:

```
// Ejemplo_02_02_09a.cs
// "for" anidados de forma incorrecta, sin llaves
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_09a
{
    public static void Main()
    {
        for (int tabla=1; tabla<=5; tabla++)
            for (int numero=1; numero<=10; numero++)
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                    tabla*numero);
                Console.WriteLine();
    }
}
```

Por eso, una alternativa recomendable es incluir siempre las llaves, aunque inicialmente esperemos repetir sólo una una orden:

```
// Ejemplo_02_02_09b.cs
// "for" anidados, variables en "for", llaves "redundantes"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_09b
{
    public static void Main()
    {
        for (int tabla=1; tabla<=5; tabla++)
        {
            for (int numero=1; numero<=10; numero++)
            {
                Console.WriteLine("{0} por {1} es {2}", tabla, numero,
                    tabla*numero);
            }

            Console.WriteLine();
        }
    }
}
```

Ejercicios propuestos:

(2.2.9.1) Crea un programa que pida un número al usuario y escriba los múltiplos de 9 que haya entre 1 ese número. Debes usar llaves en todas las estructuras de control, aunque sólo incluyan una sentencia.

(2.2.9.2) Crea un programa que pida al usuario dos números y escriba sus divisores comunes. Debes usar llaves en todas las estructuras de control, aunque sólo incluyan una sentencia.

2.2.10. Interrumpir un bucle: break

Podemos salir de un bucle antes de tiempo si lo interrumpimos con la orden **"break"**:

```
// Ejemplo_02_02_10a.cs
// "for" interrumpido con "break"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_10a
{
    public static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
            if (contador==5)
                break;

            Console.Write("{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).

Es una orden que se debe **tratar de evitar**, porque puede conducir a programas difíciles de leer, en los que no se cumple la condición de repetición del bucle, sino que se interrumpe por otros criterios. Como norma general, es preferible reescribir la condición del bucle de otra forma. En el ejemplo anterior, bastaría que la condición fuera "contador < 5". Un ejemplo ligeramente más complejo podría ser mostrar los números del 105 al 120 hasta encontrar uno que sea múltiplo de 13, que no se mostrará. Lo podríamos hacer de esta forma (poco correcta):

```
// Ejemplo_02_02_10b.cs
// "for" interrumpido con "break" (2)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_10b
{
    public static void Main()
    {
        for (int contador=105; contador<=120; contador++)
        {
            if (contador % 13 == 0)
                break;

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

O reescribirlo de esta otra (preferible):

```
// Ejemplo_02_02_10c.cs
// alternativa a un "for" interrumpido con "break"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_10c
{
    public static void Main()
    {
```

```

int contador=105;

while ( (contador<=120) && (contador % 13 != 0) )
{
    Console.Write("{0} ", contador);
    contador++;
}
}

```

(Sí, en la mayoría de los casos, un "for" se puede convertir en un "while"; veremos más detalles dentro de muy poco).

Ejercicios propuestos:

(2.2.10.1) Crea un programa que pida al usuario dos números y escriba su máximo común divisor (pista: una solución lenta pero sencilla es probar con un "for" todos los números descendiendo a partir del menor de ambos, hasta llegar a 1; cuando encuentres un número que sea divisor de ambos, interrumpes la búsqueda).

(2.2.10.2) Crea un programa que pida al usuario dos números y escriba su mínimo común múltiplo (pista: una solución lenta pero sencilla es probar con un "for" todos los números a partir del mayor de ambos, de forma creciente; cuando encuentres un número que sea múltiplo de ambos, interrumpes la búsqueda).

2.2.11. Forzar la siguiente iteración: continue

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

```

// Ejemplo_02_02_11a.cs
// "for" interrumpido con "continue"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_11a
{
    public static void Main()
    {
        for (int 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. Se podría haber usado también un "if" que escriba los valores que no sean 5, así:

```
// Ejemplo_02_02_11b.cs
// Alternativa a "for" interrumpido con "continue"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_11b
{
    public static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
            if (contador != 5)
                Console.Write("{0} ", contador);
        }
    }
}
```

Ejercicios propuestos:

(2.2.11.1) Crea un programa que escriba los números del 20 al 10, descendiendo, excepto el 13, usando "continue".

(2.2.11.2) Crea un programa que escriba los números pares del 2 al 106, excepto los que sean múltiplos de 10, usando "continue".

2.2.12. Equivalencia entre "for" y "while"

En la gran mayoría de condiciones, un bucle "for" equivale a un "while" compactado, de modo que casi cualquier "for" se puede escribir de forma alternativa como un "while", como en este ejemplo:

```
// Ejemplo_02_02_12a.cs
// "for" y "while" equivalente
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_12a
{
    public static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
            Console.Write("{0} ", contador);
        }

        Console.WriteLine();

        int n=1;
```

```

        while (n<=10)
        {
            Console.Write("{0} ", n);
            n++;
        }
    }
}

```

Incluso se comportarían igual si no se avanza de uno en uno, o se interrumpe con "break", pero no en caso de usar un "continue", como muestra este ejemplo:

```

// Ejemplo_02_02_12b.cs
// "for" y "while" equivalente... con "continue"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_02_12b
{
    public static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
            if (contador == 5)
                continue;
            Console.Write("{0} ", contador);
        }

        Console.WriteLine();

        int n=1;
        while (n<=10)
        {
            if (n == 5)
                continue;
            Console.Write("{0} ", n);
            n++;
        }
    }
}

```

En este caso, el "for" muestra todos los valores menos el 5, pero en el "while" se provoca un bucle sin fin y el programa se queda "colgado" tras escribir el número 4, porque cuando se llega al número 5, la orden "continue" hace que dicho valor no se escriba, pero que tampoco se incremente la variable, de modo que nunca se llega a pasar del 5.

Ejercicios propuestos:

(2.2.12.1) Crea un programa que escriba los números del 100 al 200, separados por un espacio, sin avanzar de línea, usando "for". En la siguiente línea, vuelve a escribirlos usando "while".

(2.2.12.2) Crea un programa que escriba los números pares del 20 al 10, descendiendo, excepto el 14, primero con "for" y luego con "while".

2.2.13. Ejercicios resueltos sobre bucles

Existen varios errores frecuentes en el manejo de los bucles. Por ejemplo, incluir un "punto" y coma tras una orden "for" o "while" puede hacer que nada se repita y que el programa se comporte de forma errónea. Por eso, aquí tienes varios ejercicios resueltos, que te ayudarán a "entrenar la vista" para localizar ese tipo de problemas:

- ¿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.Write("{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 el "for", "i" ya tiene el valor 5.

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

2.3. Saltar a otro punto del programa: 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, muy 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). Al igual que ocurría con la orden "break", será preferible replantear las condiciones de forma más natural, y no utilizar "goto".

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_02_03a.cs
// "for" y "goto"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_02_03a
{
    public static void Main()
    {
        int i, j;

        for (i=0; i<=5; i++)
            for (j=0; j<=20; j=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".

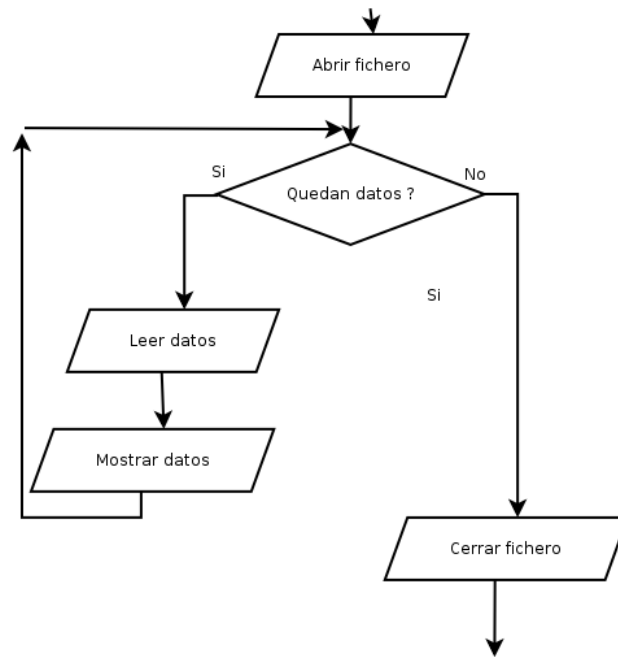
Ejercicios propuestos:

(2.3.1) Crea un programa que escriba los números del 1 al 10, separados por un espacio, sin avanzar de línea. No puedes usar "for", ni "while", ni "do..while", sólo "if" y "goto".

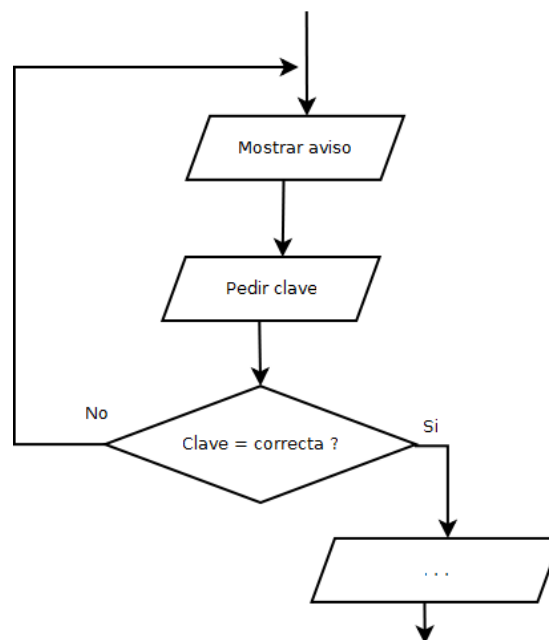
2.4. Más sobre diagramas de flujo. Diagramas de Chapin

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):



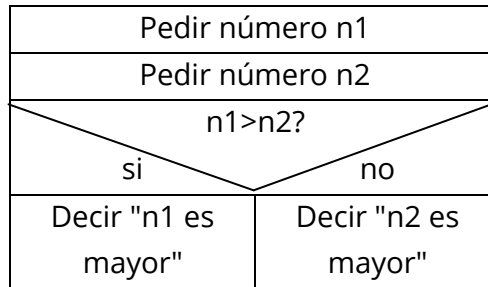
Y un "do...while" se representaría como una condición al final de un bloque que se repite:



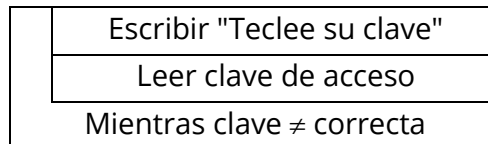
Aun así, existen otras notaciones más modernas y que pueden resultar más cómodas. Sólo comentaremos una: los diagramas de Chapin. En estos diagramas, se representa cada orden dentro de una caja:

Pedir primer número
Pedir segundo número
Mostrar primer num+segundo num

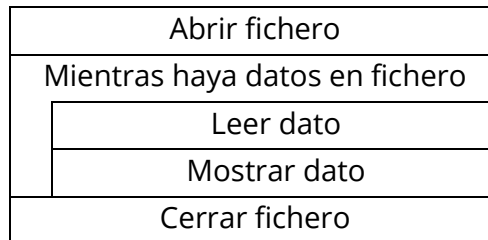
Las condiciones se denotan dividiendo las cajas en dos:



Y las condiciones repetitivas se indican dejando una barra a la izquierda, que marca qué zona es la 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".

2.5. *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 para extraer los datos de uno en uno. De momento, el único dato compuesto que hemos visto (y todavía con muy poco detalle) es la cadena de texto, "string", de la que podríamos obtener las letras una a una con "foreach" así:

```
// Ejemplo_02_05a.cs
// Primer ejemplo de "foreach"
// Introducción a C#, por Nacho Cabanes
```

```

using System;

public class Ejemplo_02_05a
{
    public static void Main()
    {
        Console.Write("Dime tu nombre: ");
        string nombre = Console.ReadLine();
        foreach(char letra in nombre)
        {
            Console.WriteLine(letra);
        }
    }
}

```

Ejercicios propuestos:

(2.5.1) Crea un programa que cuente cuantas veces aparece la letra 'a' en una frase que teclee el usuario, utilizando "foreach".

2.6. 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 quizá 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++)"). Conceptualmente, si un "for" necesita un "break" para ser interrumpido en un caso especial, es porque realmente no se trata de un contador, y en ese caso debería ser reemplazado por un "while", para que el programa resulte más legible.

Ejercicios propuestos:

(2.6.1) 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.

(2.6.2) Crear un programa que descomponga un número (que teclee el usuario) como producto de su factores primos. Por ejemplo, $60 = 2 \cdot 2 \cdot 3 \cdot 5$

(2.6.3) Crea un programa que calcule un número elevado a otro, usando multiplicaciones sucesivas.

(2.6.4) Crea un programa que "dibuje" un rectángulo formado por asteriscos, con el ancho y el alto que indique el usuario, usando dos "for" anidados. Por ejemplo, si desea anchura 4 y altura 3, el rectángulo sería así:

```
****
****
****
```

(2.6.5) Crea un programa que "dibuje" un triángulo decreciente, con la altura que indique el usuario. Por ejemplo, si el usuario dice que desea 4 caracteres de alto, el triángulo sería así:

```
****
***
**
*
```

(2.6.6) Crea un programa que "dibuje" un rectángulo hueco, cuyo borde sea una fila (o columna) de asteriscos y cuyo interior esté formado por espacios en blanco, con el ancho y el alto que indique el usuario. Por ejemplo, si desea anchura 4 y altura 3, el rectángulo sería así:

```
****
*  *
****
```

(2.6.7) Crea un programa que "dibuje" un triángulo creciente, alineado a la derecha, con la altura que indique el usuario. Por ejemplo, si el usuario dice que desea 4 caracteres de alto, el triángulo sería así:

```
  *
  **
 ***
****
```

(2.6.8) Crea un programa que devuelva el cambio de una compra, utilizando monedas (o billetes) del mayor valor posible. Supondremos que tenemos una cantidad ilimitada de monedas (o billetes) de 100, 50, 20, 10, 5, 2 y 1, y que no hay decimales. La ejecución podría ser algo como:

```
Precio? 44
Pagado? 100
Su cambio es de 56: 50 5 1
```

```
Precio? 1
Pagado? 100
Su cambio es de 99: 50 20 20 5 2 2
```

(2.6.9) Crea un programa que "dibuje" un cuadrado formado por cifras sucesivas, con el tamaño que indique el usuario, hasta un máximo de 9. Por ejemplo, si desea tamaño 5, el cuadrado sería así:

```
11111
```

22222
 33333
 44444
 55555

2.7. Una alternativa para el control errores: las excepciones

La forma "clásica" del control de errores es usar instrucciones "if", que vayan comprobando cada una de las posibles situaciones que pueden dar lugar a un error, a medida que estas situaciones llegan. Esto tiende a hacer el programa más difícil de leer, porque la lógica de la resolución del problema se ve interrumpida por órdenes que no tienen que ver con el problema en sí, sino con las posibles situaciones de error. Por eso, los lenguajes modernos, como C#, permiten una alternativa: el manejo de "excepciones".

La idea es la siguiente: "intentaremos" dar una serie de pasos, y al final de todos ellos indicaremos qué hay que hacer en caso de que alguno no se consiga completar. Esto permite que el programa sea más legible que la alternativa "convencional".

Lo haremos dividiendo el fragmento de programa en **dos bloques**:

- En un primer bloque, indicaremos los pasos que queremos "**intentar**" (try).
- A continuación, detallaremos las posibles situaciones de error (excepciones) que queremos "**interceptar**" (catch), y lo que se debe hacer en ese caso.

Lo veremos más adelante con más detalle, cuando nuestros programas sean más complejos, especialmente en el **manejo de ficheros**, pero podemos acercarnos con un primer ejemplo, que intente dividir dos números, e intercepte los posibles errores:

```
// Ejemplo_02_07a.cs
// Excepciones (1)
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_02_07a
{
```

```
    public static void Main()
    {
        int numero1, numero2, resultado;
```



```

try
{
    Console.WriteLine("Introduzca el primer numero");
    numero1 = Convert.ToInt32( Console.ReadLine() );

    Console.WriteLine("Introduzca el segundo numero");
    numero2 = Convert.ToInt32( Console.ReadLine() );

    resultado = numero1 / numero2;
    Console.WriteLine("Su división es: {0}", resultado);
}
catch (Exception errorEncontrado)
{
    Console.WriteLine("Ha habido un error: {0}",
        errorEncontrado.Message);
}
}
}

```

(La variable "errorEncontrado" es de tipo "Exception", y nos sirve para poder acceder a detalles como el mensaje correspondiente a ese tipo de excepción: errorEncontrado.Message)

En este ejemplo, si escribimos un texto en vez de un número, obtendríamos como respuesta

```

Introduzca el primer numero
hola
Ha habido un error: La cadena de entrada no tiene el formato correcto.

```

Y si el segundo número es 0, se nos diría

```

Introduzca el primer numero
3
Introduzca el segundo numero
0
Ha habido un error: Intento de dividir por cero.

```

Una alternativa más elegante es no "atrapar" todos los posibles errores a la vez, sino uno por uno (con varias sentencias "catch"), para poder tomar distintas acciones, o al menos dar mensajes de error más detallados, así:

```

// Ejemplo_02_07b.cs
// Excepciones (2)
// Introducción a C#, por Nacho Cabanes

```

```

using System;

public class Ejemplo_02_07b
{
    public static void Main()

```

```

{
    int numero1, numero2, resultado;

    try
    {
        Console.WriteLine("Introduzca el primer numero");
        numero1 = Convert.ToInt32( Console.ReadLine() );

        Console.WriteLine("Introduzca el segundo numero");
        numero2 = Convert.ToInt32( Console.ReadLine() );

        resultado = numero1 / numero2;
        Console.WriteLine("Su división es: {0}", resultado);
    }

    catch (FormatException)
    {
        Console.WriteLine("No es un número válido");
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("No se puede dividir entre cero");
    }
}
}

```

Como se ve en este ejemplo, si no vamos a usar detalles adicionales del error que ha afectado al programa, no necesitamos declarar ninguna variable de tipo `Exception`: nos basta con construcciones como `"catch (FormatException)"` en vez de `"catch (FormatException e)"`.

¿Y cómo sabemos qué excepciones debemos interceptar? La mejor forma es mirar en la "referencia oficial" para programadores de C#, la MSDN (Microsoft Developer Network): si tecleamos en un buscador de Internet algo como `"msdn convert toint32"` nos llevará a una página en la que podemos ver que hay dos excepciones que podemos obtener en ese intento de conversión de texto a entero: `FormatException` (no se ha podido convertir) y `OverflowException` (número demasiado grande). Otra alternativa más peligrosa es "probar el programa" y ver qué errores obtenemos en pantalla al introducir un valor no válido. Esta alternativa es la menos deseable, porque quizá pasemos por alto algún tipo de error que pueda surgir y que nosotros no hayamos contemplado. En cualquier caso, volveremos a las excepciones más adelante.

Ejercicios propuestos:

(2.7.1) Crea un programa que pregunte al usuario su edad y su año de nacimiento. Si la edad que introduce no es un número válido, mostrará un mensaje de aviso. Lo mismo ocurrirá si el año de nacimiento no es un número válido.

(2.7.2) Crea un programa que pregunte al usuario su edad y su año de nacimiento. Si la edad que introduce no es un número válido, mostrará un mensaje de aviso, pero aun así le preguntará su año de nacimiento.

3. Tipos de datos básicos

3.1. *Tipo de datos entero*

Hemos hablado de números enteros, de cómo realizar operaciones sencillas y de cómo usar variables para reservar espacio y así 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 hemos esquivado hasta ahora 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 nos permiten almacenar son:

Nombre	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".

```
// Ejemplo_03_01_01a.cs
// Tipos de números enteros
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_03_01_01a
{
    public static void Main()
    {
        byte edad = 74;
        ushort anyo = 2001;
        long resultado = 10000000000;
        Console.WriteLine("Los datos son {0}, {1} y {2}",
            edad, anyo, resultado);
    }
}
```

Ejercicios propuestos:

(3.1.1.1) Calcula el producto de 1.000.000 por 1.000.000, usando una variable llamada "producto", de tipo "long". Prueba también a calcularlo usando una variable de tipo "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 tenemos `ToInt16` (con signo) y `ToUInt16` (sin signo), y para los de 8 bytes existen `ToInt64` (con signo) y `ToUInt64` (sin signo).

```
// Ejemplo_03_01_02a.cs
// Conversiones para otros tipos de números enteros
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_01_02a
{
    public static void Main()
    {
        string ejemplo1 = "74";
        string ejemplo2 = "2001";
        string ejemplo3 = "10000000000";

        byte edad = Convert.ToByte(ejemplo1);
        ushort anyo = Convert.ToUInt16(ejemplo2);
        long resultado = Convert.ToInt64(ejemplo3);
        Console.WriteLine("Los datos son {0}, {1} y {2}",
            edad, anyo, resultado);
    }
}
```

Ejercicios propuestos:

(3.1.2.1) Pregunta al usuario su edad, que se guardará en un "byte". A continuación, le deberás decir que no aparenta tantos años (por ejemplo, "No aparentas 20 años").

(3.1.2.2) Pide al usuario dos números de dos cifras ("byte"), calcula su multiplicación, que se deberá guardar en un "ushort", y muestra el resultado en pantalla.

(3.1.2.3) Pide al usuario dos números enteros largos ("long") y muestra su suma, su resta y su producto.

3.1.3. Incremento y decremento

Conocemos la forma de realizar las operaciones aritméticas más habituales. Pero también existe una operación que es muy frecuente cuando se crean programas, especialmente (como ya hemos visto) a la hora de controlar bucles: incrementar el valor de una variable en una unidad:

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

Pues bien, en C# (y en otros lenguajes que derivan de C, como C++, Java y PHP), 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;
```

```
// Ejemplo_03_01_03a.cs
// Incremento y decremento
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_01_03a
{
    public static void Main()
    {
        int n = 10;
        Console.WriteLine("n vale {0}", n);
        n++;
        Console.WriteLine("Tras incrementar vale {0}", n);
        n--;
        n--;
        Console.WriteLine("Tras decrementar dos veces, vale {0}", n);
    }
}
```

Pero esto tiene algo más de dificultad de la 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:

(3.1.3.1) Crea 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.

(3.1.3.2) ¿Cuál sería el resultado de las siguientes operaciones? a=5; b=++a; c=a++; b=b*5; a=a*2; Calcúlalo a mano y luego crea un programa que lo resuelva, para ver si habías hallado la solución correcta.

3.1.4. Operaciones abreviadas: +=

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;

```
// Ejemplo_03_01_04a.cs
// Operaciones abreviadas
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_03_01_04a
{
    public static void Main()
    {
        int n = 10;
        Console.WriteLine("n vale {0}", n);
        n *= 2;
        Console.WriteLine("Tras duplicarlo, vale {0}", n);
        n /= 3;
        Console.WriteLine("Tras dividirlo entre tres, vale {0}", n);
    }
}
```

Ejercicios propuestos:

(3.1.4.1) Crea un programa que use tres variables x,y,z. Sus valores iniciales serán 15, -10, 214. Deberás 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.

(3.1.4.2) ¿Cuál sería el resultado de las siguientes operaciones? a=5; b=a+2; b-=3; c=-3; c*=2; ++c; a*=b; Crea un programa que te lo muestre.

3.1.5. Asignaciones múltiples

Ya que estamos hablando de las asignaciones, es interesante comentar que en C# es posible hacer **asignaciones múltiples**:

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

```
// Ejemplo_03_01_05a.cs
// Asignaciones múltiples
// Introducción a C#, por Nacho Cabanes
```



```

using System;

public class Ejemplo_03_01_05a
{
    public static void Main()
    {
        int a=5, b=2, c=-3;
        Console.WriteLine("a={0}, b={1}, c={2}", a, b, c);
        a = b = c = 4;
        Console.WriteLine("Ahora a={0}, b={1}, c={2}", a, b, c);
        a++; b--; c*=2;
        Console.WriteLine("Y finalmente a={0}, b={1}, c={2}", a, b, c);
    }
}

```

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). Al igual que ocurriría con los números enteros, tendremos más de un tipo de número real para elegir.

3.2.1. Coma fija y coma flotante

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 (como 003,7500), el número 970,4361 también se guardaría sin problemas, pero el 5,678642 se guardaría como 5,6786 (se perdería a partir de la cuarta cifra decimal) y el 1020 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.

Casi cualquier lenguaje de programación actual va a emplear números de coma flotante. En C# corresponden al tipo de datos llamado "float".

```

// Ejemplo_03_02_01a.cs
// Números reales (1: float)
// Introducción a C#, por Nacho Cabanes

```

```

using System;

public class Ejemplo_03_02_01a
{
    public static void Main()
    {
        int i1 = 2, i2 = 3;
        float divisionI;

        Console.WriteLine("Vamos a dividir 2 entre 3 usando enteros");
        divisionI = i1/i2;
        Console.WriteLine("El resultado es {0}", divisionI);

        float f1 = 2, f2 = 3;
        float divisionF;

        Console.WriteLine("Vamos a dividir 2 entre 3 usando reales");
        divisionF = f1/f2;
        Console.WriteLine("El resultado es {0}", divisionF);
    }
}

```

Usando "float" sí hemos podido dividir con decimales. El resultado de este programa es:

```

Vamos a dividir 2 entre 3 usando enteros
El resultado es 0
Vamos a dividir 2 entre 3 usando reales
El resultado es 0,6666667

```

Ejercicios propuestos:

(3.2.1.1) Crea un programa que muestre el resultado de dividir 3 entre 4 usando números enteros y luego usando números de coma flotante.

(3.2.1.2) ¿Cuál sería el resultado de las siguientes operaciones, usando números reales? `a=5; a/=2; a+=1; a*=3; --a;`

3.2.2. Simple y doble precisión

En la mayoría de lenguajes de programación, contamos con dos tamaños de números reales 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") usaremos 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, el tipo "decimal", que se acerca a las 30 cifras significativas:

	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

Así, podríamos plantear el ejemplo anterior con un "double" para obtener un resultado más preciso:

```
// Ejemplo_03_02_02a.cs
// Números reales (2: double)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_02a
{
    public static void Main()
    {
        double n1 = 2, n2 = 3;
        double division;

        Console.WriteLine("Vamos a dividir 2 entre 3");
        division = n1/n2;
        Console.WriteLine("El resultado es {0}", division);
    }
}
```

Ahora su resultado sería:

```
Vamos a dividir 2 entre 3
El resultado es 0,666666666666667
```

Si queremos **dar un valor** inicial a un dato "float", debemos llevar cuidado. Es válido darle un valor entero, como hemos hecho en el ejemplo anterior:

```
float x = 2;
```

pero no podremos dar un valor que contenga cifras decimales, porque el compilador mostrará un mensaje de error diciendo que lo que aparece a la derecha es para él un dato "double" y el tipo de la variable es "float", así que se perderá precisión:

```
float pi = 3.14; // No válido
```

Hay un par de formas de solucionarlo. La más sencilla es añadir el sufijo "f" al número, para indicar al compilar que debe ser tratado como un "float":

```
float pi = 3.14f; // Dato correcto
```

Otra forma alternativa es forzar una "conversión de tipos", como veremos dentro de muy poco.

Así, podemos crear un programa que pida al usuario el radio de una circunferencia (que será un número entero) para mostrar la longitud de la circunferencia (cuyo valor será $2 * \pi * \text{radio}$) podría ser:

```
// Ejemplo_03_02_02a.cs
// Números reales: valor inicial de un float
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_02b
{
    public static void Main()
    {
        int radio;
        float pi = 3.14f; // Atención a la "f" del final

        Console.WriteLine("Introduce el radio");
        radio = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("La longitud de la circunferencia es");
        Console.WriteLine(2 * pi * radio);
    }
}
```

Ejercicios propuestos:

(3.2.2.1) Crea un programa que muestre el resultado de dividir 13 entre 6 usando números enteros, luego usando números de coma flotante de simple precisión y luego con números de doble precisión.

(3.2.2.2) Calcula el área de un círculo, dado su radio, que será un número entero (área = $\pi * \text{radio al cuadrado}$)

3.2.3. Pedir números reales al usuario

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_03_02_03a.cs
```

```
// Números reales: pedir al usuario
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_03a
{
    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, cuando se pidan datos al usuario, parte del trabajo se le encarga al sistema operativo, de modo que si éste sabe que en nuestro país se usa la "coma" para separar los decimales, considerará que la coma es el separador correcto y no el punto, que será ignorado. Por ejemplo, ocurre si introducimos los datos 23,6 y 34.2 en la versión española de Windows 8 obtendremos como respuesta:

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

Ejercicios propuestos:

(3.2.3.1) Calcula el volumen de una esfera, dado su radio, que será un número de doble precisión (volumen = $\pi * \text{radio al cubo} * 4/3$)

(3.2.3.2) Crea un programa que pida al usuario a una distancia (en metros) y el tiempo necesario para recorrerla (como tres números: horas, minutos, segundos), y muestre la velocidad, en metros por segundo, en kilómetros por hora y en millas por hora (pista: 1 milla = 1.609 metros).

(3.2.3.3) Halla las soluciones de una ecuación de segundo grado del tipo $y = Ax^2 + Bx + C$. Pista: la raíz cuadrada de un número x se calcula con `Math.Sqrt(x)`

(3.2.3.4) Si se ingresan E euros en el banco a un cierto interés I durante N años, el dinero obtenido viene dado por la fórmula del interés compuesto: Resultado = e

$(1+i)^n$ Aplicarlo para calcular en cuanto se convierten 1.000 euros al cabo de 10 años al 3% de interés anual.

(3.2.3.5) Crea un programa que muestre los primeros 20 valores de la función $y = x^2 - 1$

(3.2.3.6) Crea un programa que "dibuje" la gráfica de $y = (x-5)^2$ para valores de x entre 1 y 10. Deberá hacerlo dibujando varios espacios en pantalla y luego un asterisco. La cantidad de espacios dependerá del valor obtenido para "y".

(3.2.3.7) Escribe un programa que calcule una aproximación de PI mediante la expresión: $\pi/4 = 1/1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 \dots$ El usuario deberá indicar la cantidad de términos a utilizar, y el programa mostrará todos los resultados hasta esa cantidad de términos. Debes hacer todas las operaciones con "double".

3.2.4. Conversión de tipos (typecast)

Cuando queremos convertir de un tipo de número a otro (por ejemplo, para quedarnos con la parte entera de un número real), tenemos dos alternativas:

- Usar Convert, como en `x = Convert.ToInt32(y);`
- Hacer un **forzado de tipos**, que es más rápido pero no siempre es posible, sólo cuando el tipo de origen y el de destino se parecen lo suficiente. Para ello, se precede el valor de la variable con el nuevo tipo de datos entre paréntesis, así: `x = (int) y;`

Por ejemplo, podríamos retocar el programa que calculaba la longitud de la circunferencia, de modo que su resultado sea un "double", que luego convertiremos a "float" y a "int" forzando el nuevo tipo de datos:

```
// Ejemplo_03_02_04a.cs
// Números reales: typecast
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_04a
{
    public static void Main()
    {
        double radio;
        float pi = (float) 3.141592654;
        double longitud;
        float longitudSimplePrec;
        int longitudEntera;

        Console.WriteLine("Introduce el radio");
        radio = Convert.ToDouble(Console.ReadLine());

        longitud = 2 * pi * radio;
```

```

Console.WriteLine("La longitud de la circunferencia es");
Console.WriteLine(longitud);

longitudSimplePrec = (float) longitud;
Console.WriteLine("Y con simple precisión");
Console.WriteLine(longitudSimplePrec);

longitudEntera = (int) longitud;
Console.WriteLine("Y como número entero");
Console.WriteLine(longitudEntera);
    }
}

```

Su resultado sería:

```

Introduce el radio
2,3456789
La longitud de la circunferencia es
14,7383356099727
Y con simple precisión
14,73834
Y como número entero
14

```

Ejercicios propuestos:

(3.2.4.1) Crea un programa que calcule la raíz cuadrada del número que introduzca el usuario. La raíz se deberá calcular como "double", pero el resultado se mostrará como "float"

(3.2.4.2) Crea una nueva versión del un programa que calcula una aproximación de PI mediante la expresión: $\pi/4 = 1/1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 \dots$ con tantos términos como indique el usuario. Debes hacer todas las operacion con "double", pero mostrar el resultado como "float".

3.2.5. Formatear números

En más de una ocasión nos interesará afinar la apariencia de los números en pantalla, para mostrar sólo 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, o con sólo un decimal.

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")`

Algunos 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_03_02_05a.cs
// Formato de números reales
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_05a
{
    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:

(3.2.5.1) 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.

(3.2.5.2) Crea un programa que use tres variables x,y,z. Las tres serán números reales, y nos bastará con dos cifras decimales. Se 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).

(3.2.5.3) Calcula el perímetro, área y diagonal de un rectángulo, a partir de su ancho y alto (perímetro = suma de los cuatro lados; área = base x altura; diagonal, usando el teorema de Pitágoras). Muestra todos ellos con una cifra decimal.

(3.2.5.4) Calcula la superficie y el volumen de una esfera, a partir de su radio (superficie = $4 * \pi * \text{radio al cuadrado}$; volumen = $4/3 * \pi * \text{radio al cubo}$). Usa datos "doble" y muestra los resultados con 5 cifras decimales.

3.2.6. Cambios de base

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_03_02_06a.cs
// De decimal a hexadecimal y binario
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_06a
{
    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)

Ejercicios propuestos:

(3.2.6.1) Crea un programa que pida números (en base 10) al usuario y muestre su equivalente en sistema binario y en hexadecimal. Debe repetirse hasta que el usuario introduzca el número 0.

(3.2.6.2) Crea un programa que pida al usuario la cantidad de rojo (por ejemplo, 255), verde (por ejemplo, 160) y azul (por ejemplo, 0) que tiene un color, y que muestre ese color RGB en notación hexadecimal (por ejemplo, FFA000).

(3.2.6.3) Crea un programa para mostrar los números del 0 a 255 en hexadecimal, en 16 filas de 16 columnas cada una (la primera fila contendrá los números del 0 al 15 –decimal-, la segunda del 16 al 31 –decimal- y así sucesivamente).

Para convertir en sentido contrario, de **hexadecimal o binario a decimal**, podemos usar `Convert.ToInt32`, como se ve en el siguiente ejemplo. Es importante destacar que una constante hexadecimal se puede expresar precedida por **"0x"**, como en `"int n1 = 0x13;"` (donde n1 tendría el valor 16+3=19, expresado en base 10). En los lenguajes C y C++, un valor precedido por **"0"** se considera **octal**, de modo que para `"int n2 = 013;"` el valor decimal de n2 sería 8+3=11, pero en C# no es así: un número que empiece por 0 (no por 0x) se considera que está escrito en base 10, como se ve en este ejemplo:

```
// Ejemplo_03_02_06b.cs
// De hexadecimal y binario a decimal
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_06b
{
    public static void Main()
    {
        int n1 = 0x13;
        int n2 = Convert.ToInt32("1a", 16);

        int n3 = 013; // No es octal, al contrario que en C y C++
        int n4 = Convert.ToInt32("14", 8);

        int n5 = Convert.ToInt32("11001001", 2);

        Console.WriteLine( "{0} {1} {2} {3} {4}",
                           n1, n2, n3, n4, n5);
    }
}
```

Que mostraría:

```
19 26 13 12 201
```

Ejercicios propuestos:

(3.2.6.4) Crea un programa que pida números binarios al usuario y muestre su equivalente en sistema hexadecimal y en decimal. Debe repetirse hasta que el usuario introduzca la palabra "fin".

3.3. Tipo de datos carácter

3.3.1. Leer y mostrar caracteres

Como ya vimos brevemente, en C# 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 (que debería tener sólo una letra) 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_03_03_01a.cs
// Tipo de datos "char"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_03_01a
{
    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);
    }
}

```

Ejercicios propuestos

(3.3.1.1) Crea un programa que pida una letra al usuario y diga si se trata de una vocal.

(3.3.1.2) Crea un programa que muestre una de cada dos letras entre la que teclee el usuario y la "z". Por ejemplo, si el usuario introduce una "a", se escribirá "aceg...".

(3.3.1.3) Crea un programa que pida al usuario el ancho (por ejemplo, 4) y el alto (por ejemplo, 3) y una letra (por ejemplo, X) y escriba un rectángulo formado por esa cantidad de letras:

```

XXXX
XXXX
XXXX

```

3.3.2. 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**, con `\'` unas **comillas simples**, con `\\` se escribe una barra invertida y con `\n` se avanzará a la línea siguiente de pantalla (es preferible evitar este último, porque puede no funcionar correctamente en todos los sistemas operativos; más adelante veremos una alternativa más segura).

Estas secuencias especiales son las siguientes:

Secuencia	Significado
<code>\a</code>	Emite un pitido
<code>\b</code>	Retroceso (permite borrar el último carácter)
<code>\f</code>	Avance de página (expulsa una hoja en la impresora)
<code>\n</code>	Avanza de línea (salta a la línea siguiente)
<code>\r</code>	Retorno de carro (va al principio de la línea)
<code>\t</code>	Salto de tabulación horizontal
<code>\v</code>	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_03_03_02a.cs
// Secuencias de escape
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_03_02a
{
    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 al estilo de MsDos y Windows, deberíamos duplicar todas las barras invertidas: c:\\datos\\ejemplos\\curso\\ejemplo1. En este caso, se puede usar una **arroba** (@) antes del texto, en vez de usar 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

(3.3.2.1) Crea 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

Al contrario que en lenguajes más antiguos (como C), las cadenas de texto en C# son tan fáciles de manejar como los demás tipos de datos que hemos visto. Los detalles que hay que tener en cuenta en un primer acercamiento son:

- 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.
- Podemos comparar su valor usando "==" (igualdad) o "!=" (desigualdad).

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_03_04a.cs
// Uso basico de "string"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_04a
{
    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);

        if (frase == "Hola!")
            Console.WriteLine("Hola a ti también! ");
    }
}
```

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, en el próximo tema.

Ejercicios propuestos:

(3.4.1) 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.

(3.4.2) Crear un programa que pida al usuario un nombre y una contraseña. La contraseña se debe introducir dos veces. Si las dos contraseñas no son iguales, se avisará al usuario y se le volverán a pedir las dos contraseñas.

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:

```
// Ejemplo básico
partidaTerminada = false;
if (vidas == 0) partidaTerminada = true;
// Notación alternativa, sin usar "if"
partidaTerminada = vidas == 0;

// Ejemplo más desarrollado
if (enemigos == 0 && (nivel == ultimoNivel))
    partidaTerminada = true;
else
    partidaTerminada = false;
// Notación alternativa, sin usar "if"
partidaTerminada = (enemigos == 0 && (nivel == ultimoNivel));
```

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

```
// Ejemplo_03_05a.cs
// Variables bool
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_05a
{
    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 símbolo.");
    }
}
```

Ejercicios propuestos:

(3.5.1) Crea 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.

(3.5.2) Crea una versión alternativa del ejercicio 3.5.1, que use "if" en vez del operador condicional.

(3.5.3) Crea un programa que use el operador condicional para dar a una variable llamada "ambosPares" (booleana) el valor "true" si dos números introducidos por el usuario son pares, o "false" si alguno es impar.

(3.5.4) Crea una versión alternativa del ejercicio 3.5.3, que use "if" en vez del operador condicional.

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, y a los que accederemos usando el mismo nombre.

Por ejemplo, si queremos definir un grupo de números enteros, el tipo de datos que usaremos para declararlo será "int []":

```
int[] ejemplo;
```

Cuando sepamos cuantos datos vamos a guardar (por ejemplo 4), podremos reservar espacio con la orden "new", así:

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

Si sabemos el tamaño desde el principio, podemos reservar espacio a la vez que declaramos la variable:

```
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]. Por tanto, podríamos dar el valor 15 al primer elemento de nuestro array así:

```
ejemplo[0] = 15;
```

Habitualmente, como un array representa un conjunto de números, utilizaremos nombres en plural, como en

```
int[] datos = new int[100];
```

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

```
// Ejemplo_04_01_01a.cs
// Primer ejemplo de tablas (arrays)
// Introducción a C#, por Nacho Cabanes
```

```

using System;

public class Ejemplo_04_01_01a
{
    public static void Main()
    {
        int[] numeros = new int[5]; // Un array de 5 números enteros
        int suma; // Un entero que será la suma

        numeros[0] = 200; // Les damos valores
        numeros[1] = 150;
        numeros[2] = 100;
        numeros[3] = -50;
        numeros[4] = 300;
        suma = numeros[0] + // Y calculamos la suma
            numeros[1] + numeros[2] + numeros[3] + numeros[4];
        Console.WriteLine("Su suma es {0}", suma);
        // Nota: esta es la forma más ineficiente e incómoda
        // Lo mejoraremos...
    }
}

```

Ejercicios propuestos:

(4.1.1.1) Un programa que pida al usuario 4 números, los memorice (utilizando un array), calcule su media aritmética y después muestre en pantalla la media y los datos tecleados.

(4.1.1.2) Un programa que pida al usuario 5 números reales (pista: necesitarás un array de "float") y luego los muestre en el orden contrario al que se introdujeron.

4.1.2. Valor inicial de un array

Al igual que ocurría con las variables "normales", podemos dar valor inicial a los elementos de una tabla al principio del programa, si conocemos todos su valores. En este caso, los indicaremos todos entre llaves, separados por comas:

```

// Ejemplo_04_01_02a.cs
// Segundo ejemplo de tablas: valores iniciales con []
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_01_02a
{
    public static void Main()
    {
        int[] numeros = // Un array de 5 números enteros
            {200, 150, 100, -50, 300};
        int suma; // Un entero que será su suma

        suma = numeros[0] + // Hallamos la suma
            numeros[1] + numeros[2] + numeros[3] + numeros[4];
        Console.WriteLine("Su suma es {0}", suma);
    }
}

```

```

        // Nota: esta forma es algo menos engorrosa, pero todavía no
        // está bien hecho. Lo seguiremos mejorando.
    }
}

```

Nota: el formato completo para la declaración de un array con valores iniciales es éste. incluyendo tanto la orden "new" como los valores entre llaves:

```
int[] numeros = new int[5] {200, 150, 100, -50, 300};
```

Pero, como se ve en el ejemplo anterior, casi siempre se podrá abreviar, y no será necesario usar "new" junto con el tamaño, ya que el compilador lo puede deducir al analizar el contenido del array:

```
int[] numeros = {200, 150, 100, -50, 300};
```

Ejercicios propuestos:

(4.1.2.1) 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.

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 = numeros[0] + numeros[1] + numeros[2] + numeros[3] + numeros[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: 0
for (int i=0; i<=4; i++) // Y calculamos la suma repetitiva
    suma += numeros[i];

```

El fuente completo podría ser así:

```

// Ejemplo_04_01_03a.cs
// Tercer ejemplo de tablas: valores iniciales con llaves
// y recorrido con "for"
// Introducción a C#, por Nacho Cabanes

```

```
using System;
```

```

public class Ejemplo_04_01_03a
{
    public static void Main()
    {
        int[] numeros =           // Un array de 5 números enteros
            {200, 150, 100, -50, 300};
        int suma;                 // Un entero que será su suma

        suma = 0;                 // Valor inicial de la suma: 0
        for (int i=0; i<=4; i++)  // Y calculamos la suma repetitiva
            suma += numeros[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 sería evidente.

Lógicamente, también podemos **pedir los datos** al usuario de forma repetitiva, usando un "for", "while" o "do..while":

```

// Ejemplo_04_01_03b.cs
// Cuarto ejemplo de tablas: introducir datos repetitivos
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_01_03b
{
    public static void Main()
    {
        int[] numeros = new int[5];
        int suma;

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

        suma = 0;                 // Y calculamos la suma
        for (int i=0; i<=4; i++)
            suma += numeros[i];

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

```

Ejercicios propuestos:

(4.1.3.1) Crea un programa que pida al usuario 6 números enteros cortos y luego los muestre en orden inverso (pista: usa un array para almacenarlos y "for" para mostrarlos).

(4.1.3.2) Crea un programa que pregunte al usuario cuántos números enteros va a introducir (por ejemplo, 10), le pida todos esos números, los guarde en un array y finalmente calcule y muestre la media de esos números.

(4.1.3.3) Un programa que pida al usuario 10 reales de doble precisión, calcule su media y luego muestre los que están por encima de la media.

(4.1.3.4) Un programa que almacene en una tabla el número de días que tiene cada mes (de un 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).

(4.1.3.5) A partir del ejercicio anterior, crea otro que pida al usuario que le indique la fecha, formada por día (1 al 31) y el mes (1=enero, 12=diciembre), y como respuesta muestre en pantalla el número de días que quedan hasta final de año.

(4.1.3.6) Un programa que pida 10 nombres y los memorice (pista: esta vez se trata de un array de "string"). Después 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".

(4.1.3.7) Un programa que prepare espacio para guardar 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.

(4.1.3.8) Un programa que reserve espacio para un vector de 3 componentes, pida al usuario valores para dichas componentes (por ejemplo [2, -5, 7]) y muestre su módulo (raíz cuadrada de la suma de sus componentes al cuadrado).

(4.1.3.9) Un programa que reserve espacio para dos vectores de 3 componentes, pida al usuario sus valores y calcule la suma de ambos vectores (su primera componente será x_1+y_1 , la segunda será x_2+y_2 y así sucesivamente).

(4.1.3.10) Un programa que reserve espacio para dos vectores de 3 componentes, pida al usuario sus valores y calcule su producto escalar ($x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$).

(4.1.3.11) Un programa que pida al usuario 4 números enteros y calcule (y muestre) cuál es el mayor de ellos. Nota: para calcular el mayor valor de un array, hay que comparar cada uno de los valores que tiene almacenados el array con el que hasta ese momento es el máximo provisional. El valor inicial de este máximo provisional no debería ser cero (porque el resultado sería incorrecto si todos los números son negativos), sino el primer elemento del array.

4.1.4. Operaciones habituales con arrays: buscar, añadir, insertar, borrar

Algunas operaciones con datos pertenecientes a un array son especialmente frecuentes: buscar si existe un cierto dato, añadir un dato al final de los existentes, insertar un dato entre dos que ya hay, borrar uno de los datos almacenados, etc. Por eso, vamos a ver las pautas básicas para realizar estas operaciones, y un fuente de ejemplo.

Para ver **si un dato existe**, habrá que recorrer todo el array, comparando el valor almacenado con el dato que se busca. Puede interesarnos simplemente saber si está o no (con lo que se podría interrumpir la búsqueda en cuanto aparezca una primera vez) o ver en qué posiciones se encuentra (para lo que habría que recorrer todo el array). Si el array estuviera ordenado, se podría buscar de una forma más rápida, pero la veremos más adelante.

Para encontrar **el máximo o el mínimo** de los datos, tomaremos el primero de los datos como valor provisional, y compararemos con cada uno de los demás, para ver si está por encima o debajo de ese máximo o mínimo provisional, y cambiarlo si fuera necesario.

Para poder **añadir** un dato al final de los ya existentes, necesitamos que el array no esté completamente lleno, y llevar un contador de cuántas posiciones hay ocupadas, de modo que seamos capaces de guardar el dato en la primera posición libre.

Para **insertar** un dato en una cierta posición, los que queden detrás deberán desplazarse "hacia la derecha" para dejarle hueco. Este movimiento debe empezar desde el final para que cada dato que se mueve no destruya el que estaba a continuación de él. También habrá que actualizar el contador, para indicar que queda una posición libre menos.

Si se quiere **borrar** el dato que hay en una cierta posición, los que estaban a continuación deberán desplazarse "hacia la izquierda" para que no queden huecos. Como en el caso anterior, habrá que actualizar el contador, pero ahora para indicar que queda una posición libre más.

Vamos a verlo con un ejemplo:

```
// Ejemplo_04_01_04a.cs
```

```
// Añadir y borrar en un array
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_01_04a
{
    public static void Main()
    {
        int[] datos = {10, 15, 12, 0, 0};

        int capacidad = 5;           // Capacidad maxima del array
        int cantidad = 3;             // Número real de datos guardados

        int i;                       // Para recorrer los elementos

        // Mostramos el array
        for (i=0; i<cantidad; i++)
            Console.Write("{0} ",datos[i]);
        Console.WriteLine();

        // Buscamos el dato "15"
        for (i=0; i<cantidad; i++)
            if (datos[i] == 15)
                Console.WriteLine("15 encontrado en la posición {0} ", i+1);

        // Buscamos el máximo
        int maximo = datos[0];
        for (i=1; i<cantidad; i++)
            if (datos[i] > maximo)
                maximo = datos[i];
        Console.WriteLine("El máximo es {0} ", maximo);

        // Añadimos un dato al final
        Console.WriteLine("Añadiendo 6 al final");
        if (cantidad < capacidad)
        {
            datos[cantidad] = 6;
            cantidad++;
        }

        // Y volvemos a mostrar el array
        for (i=0; i<cantidad; i++)
            Console.Write("{0} ",datos[i]);
        Console.WriteLine();

        // Borramos el segundo dato
        Console.WriteLine("Borrando el segundo dato");
        int posicionBorrar = 1;
        for (i=posicionBorrar; i<cantidad-1; i++)
            datos[i] = datos[i+1];
        cantidad--;

        // Y volvemos a mostrar el array
        for (i=0; i<cantidad; i++)
            Console.Write("{0} ",datos[i]);
        Console.WriteLine();

        // Insertamos 30 en la tercera posición
        if (cantidad < capacidad)
```

```

    {
        Console.WriteLine("Insertando 30 en la posición 3");
        int posicionInsertar = 2;
        for (i=cantidad; i>posicionInsertar; i--)
            datos[i] = datos[i-1];
        datos[posicionInsertar] = 30;
        cantidad++;
    }

    // Y volvemos a mostrar el array
    for (i=0; i<cantidad; i++)
        Console.Write("{0} ",datos[i]);
    Console.WriteLine();
}
}

```

que tendría como resultado:

```

10 15 12
15 encontrado en la posición 2
El máximo es 15
Añadiendo 6 al final
10 1512 6
Borrando el segundo dato
10 12 6
Insertando 30 en la posición 3
10 12 30 6

```

Este programa "no dice nada" cuando no se encuentra el dato que se está buscando. Se puede mejorar usando una variable "booleana" que nos sirva de testigo, de forma que al final nos avise si el dato no existía (no sirve emplear un "else", porque en cada pasada del bucle "for" no sabemos si el dato no existe, sólo sabemos que no está en la posición actual).

Ejercicios propuestos:

(4.1.4.1) Crea una variante del ejemplo anterior (04_01_04a) que pida al usuario el dato a buscar, avise si ese dato no aparece, y que diga cuántas veces se ha encontrado en caso contrario.

(4.1.4.2) Crea una variante del ejemplo anterior (04_01_04a) que añada un dato introducido por el usuario al final de los datos existentes.

(4.1.4.3) Crea una variante del ejemplo anterior (04_01_04a) que inserte un dato introducido por el usuario en la posición que elija el usuario. Debe avisar si la posición escogida es incorrecta (porque esté más allá del final de los datos).

(4.1.4.4) Crea una variante del ejemplo anterior (04_01_04a) que borre el dato que se encuentre en la posición que elija el usuario. Debe avisar si la posición escogida no es válida.

(4.1.4.5) Crea un programa que prepare 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 (como ya se ha comentado, los que queden 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 (como se ha visto, 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.

4.1.5. Constantes

En ocasiones, manejaremos valores que realmente no van a variar. Es el caso del tamaño máximo de un array. En esos casos, por legibilidad y por facilidad de mantenimiento del programa, puede ser preferible no usar el valor numérico, sino una variable. Dado que el valor de ésta no debería cambiar, podemos usar la palabra "const" para indicar que debe ser constante, y el compilador no permitirá que la modifiquemos por error. Además, por convenio, para que sea fácil distinguir una constante de una variable, se suele escribir su nombre totalmente en mayúsculas:

```
const int MAXIMO = 5;
```

Así, una nueva versión del fuente del apartado 4.1.3 (b), usando una constante para la capacidad del array podría ser:

```
// Ejemplo_04_01_05a.cs
// Quinto ejemplo de tablas: constantes
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_01_05a
{
    public static void Main()
    {
        const int MAXIMO = 5;           // Cantidad de datos
```

```

int[] numeros = new int[MAXIMO];
int suma;

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

suma = 0;                      // Y calculamos la suma
for (int i=0; i<MAXIMO; i++)
    suma += numeros[i];

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

```

Ejercicios propuestos:

(4.1.5.1) Crea un programa que contenga un array con los nombres de los meses del año. El usuario podrá elegir entre verlos en orden natural (de Enero a Diciembre) o en orden inverso (de Diciembre a Enero). Usa constantes para el valor máximo del array en ambos recorridos.

4.2. Arrays bidimensionales

Podemos declarar tablas de **dos o más dimensiones**. Por ejemplo, si un profesor quiere guardar datos de dos grupos de alumnos, cada uno de los cuales tiene 20 personas, tendría dos opciones:

- Se puede 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 de uso con **arrays "rectangulares"**, de la forma [2,20], lo que podríamos llamar el "estilo Pascal". En este ejemplo usaremos tanto arrays con valores prefijados, como arrays para los que reservemos espacio con "new" y a los que demos valores en un segundo paso:

```
// Ejemplo_04_02a.cs
// Array de dos dimensiones "rectangulares" (estilo Pascal)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_02a
{
    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, prefijados
        {
            {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. Si ya has estudiado la teoría de matrices, más adelante tienes algunos ejercicios propuestos para aplicar esos conocimientos al uso de arrays bidimensionales.

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_04_02b.cs
// Array de arrays (array de dos dimensiones al estilo C)
// Introducción a C#, por Nacho Cabanes
```

```

using System;

public class Ejemplo_04_02b
{
    public static void Main()
    {
        int[][] notas; // Array de dos dimensiones
        notas = new int[3][]; // Seran 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 ultimo

        // 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();
        }
    } // Fin de "Main"
}

```

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:

(4.2.1) Un programa que pida 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.

(4.2.2) Un programa que pida 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.

(4.2.3) Un programa que pregunte al usuario el tamaño que tendrán dos bloques de números enteros (por ejemplo, uno de 10 elementos y otro de 12). Luego

pedirá los datos para ambos bloques de datos. Finalmente deberá mostrar el mayor dato que se ha introducido en cada uno de ellos.

Si has estudiado **álgebra de matrices**:

(4.2.4) Un programa que calcule el determinante de una matriz de 2x2.

(4.2.5) Un programa que calcule el determinante de una matriz de 3x3.

(4.2.6) Un programa que calcule si las filas de una matriz son linealmente dependientes.

(4.2.7) Un programa que use matrices para resolver un sistema de ecuaciones lineales mediante el método de Gauss.

4.3. Estructuras o registros

4.3.1. Definición y acceso a los datos

Un **registro** es una agrupación de datos, llamados **campos**, los cuales no necesariamente son del mismo tipo. Se definen con la palabra "**struct**".

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 siempre 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_04_03_01a.cs
// Registros (struct)
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_04_03_01a
{
    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);
    }
}

```

Ejercicios propuestos:

(4.3.1.1) Crea 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 structs

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_04_03_02a.cs
// Array de struct
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_03_02a
{
    struct tipoPersona
    {
        public string nombre;
        public char inicial;
        public int edad;
        public float nota;
    }

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

        personas[0].nombre = "Juan";
        personas[0].inicial = 'J';
        personas[0].edad = 20;
        personas[0].nota = 7.5f;
        Console.WriteLine("La edad de {0} es {1}",

```

```

        personas[0].nombre, personas[0].edad);

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

```

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

(4.3.2.1) Amplia el programa del ejercicio 4.3.1.1, 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).

(4.3.2.2) Crea 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. structs anidados

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_04_03_03a.cs
// Registros anidados
// Introducción a C#, por Nacho Cabanes

```

```

using System;

public class Ejemplo_04_03_03a
{
    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:

(4.3.3.1) Amplia el programa 4.3.2.1, 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 (=) y de comparar cual es su valor (==).

Vamos a repasar todas esas posibilidades, junto con la de formar una cadena a partir de otras si las unimos con el símbolo de la suma (+), lo que llamaremos "**concatenar**" cadenas. Un ejemplo que nos pidiese nuestro nombre y nos saludase usando todo ello podría ser:


```
// Ejemplo_04_04_01a.cs
// Cadenas de texto (1: manejo básico)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_04_01a
{
    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);
    }
}
```

Ejercicios propuestos:

(4.4.1.1) Crea un programa que te pida tu nombre y lo escriba 5 veces.

(4.4.1.2) Crea un programa que pida al usuario su nombre. Si se llama como tú (por ejemplo, "Nacho"), responderá "Bienvenido, jefe". En caso contrario, le saludará por su nombre.

(4.4.1.3) 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".

(4.4.1.4) Crea un programa que pida al usuario dos números enteros y después una operación que realizar con ellos. La operación podrá ser "suma", "resta", "multiplicación" y "división", que también se podrán escribir de forma abreviado con los operadores matemáticos "+", "-", "*" y "/". Para multiplicar también se podrá usar una "x", minúscula o mayúscula. A continuación se mostrará el resultado de esa operación (por ejemplo, si los números son 3 y 6 y la operación es "suma", el resultado sería 9). La operación debes tomarla como una cadena de texto y analizarla con un "switch".

4.4.2. Cómo acceder a las letras que forman una cadena

Podemos leer 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.

```
// Ejemplo_04_04_02a.cs
// Cadenas de texto (2: acceder a una letra)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_04_02a
{
    public static void Main()
    {
        string saludo = "Hola";
        Console.WriteLine("La tercera letra de {0} es {1}",
            saludo, saludo[2]);
    }
}
```

Ejercicios propuestos:

(4.4.2.1) Crea un programa que pregunte al usuario su nombre y le responda cuál es su inicial.

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".

```
// Ejemplo_04_04_03a.cs
// Cadenas de texto (3: longitud)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_04_03a
{
    public static void Main()
    {
        string saludo = "Hola";
        int longitud = saludo.Length;
        Console.WriteLine("La longitud de {0} es {1}", saludo, longitud);
        for (int i=0; i<longitud; i++)
```

```

    {
        Console.WriteLine("La letra {0} es {1}", i, saludo[i]);
    }
}

```

Ejercicios propuestos:

(4.4.3.1) 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".

(4.4.3.2) Un programa que pida una frase al usuario y la muestre en orden inverso (de la última letra a la primera).

(4.4.3.3) Un programa que pida al usuario una frase, después una letra y finalmente diga si aparece esa letra como parte de esa frase o no.

(4.4.3.4) 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.

(4.4.3.5) 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.

```

// Ejemplo_04_04_04a.cs
// Cadenas de texto (4: substring)
// Introducción a C#, por Nacho Cabanes

```

```
using System;
```

```

public class Ejemplo_04_04_04a
{
    public static void Main()
    {
        string saludo = "Hola";
        string subcadena = saludo.Substring(1,3);
        Console.WriteLine("Una subcadena de {0} es {1}",
            saludo, subcadena);
    }
}

```

```
}
```

Ejercicios propuestos:

(4.4.4.1) Un programa que te pida tu nombre y lo muestre en pantalla separando cada letra de la siguiente con un espacio, similar al 4.4.3.1, pero esta vez usando "Substring". Por ejemplo, si tu nombre es "Juan", debería aparecer en pantalla "J u a n".

(4.4.4.2) Un programa que te pida tu nombre y lo muestre en pantalla como un triángulo creciente. Por ejemplo, si tu nombre es "Juan", debería aparecer en pantalla:

```
J
Ju
Jua
Juan
```

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, siendo 0 la primera posición (o devuelve el valor -1, si no aparece):

```
if (nombre.IndexOf("Juan") >= 0) Console.Write("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).

Si solamente queremos ver si aparece, pero no nos importa en qué posición está, nos bastará con usar "Contains":

```
if (nombre.Contains("Juan")) ...
```

Un ejemplo de la utilización de IndexOf podría ser:

```
// Ejemplo_04_04_05a.cs
// Cadenas de texto (5: substring)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_04_05a
{
    public static void Main()
    {
        string saludo = "Hola";
        string subcadena = "ola";
        Console.WriteLine("{0} aparece dentro de {1} en la posición {2}",
            subcadena, saludo, saludo.IndexOf(subcadena) );
    }
}
```

Ejercicios propuestos:

(4.4.5.1) Un programa que pida al usuario 10 frases, las guarde en un array, y luego le pregunte textos de forma repetitiva, e indique si cada uno de esos textos aparece como parte de alguno de los elementos del array. Terminará cuando el texto introducido sea "fin".

(4.4.5.2) Crea una versión del ejercicio 4.4.5.1 en la que, en caso de que alguno de los textos aparezca como subcadena, se avise además si se encuentra exactamente al principio.

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_04_04_06a.cs
// Cadenas de texto (6: manipulaciones diversas)
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_04_06a
{
    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 posición de "que" es: 6
 La última A está en la posición: 17
 En mayúsculas: HOLA, QUE TAL ESTAS
 En minúsculas: hola, que tal estas
 Si insertamos ", tío": Hola, tío, que tal estas
 Si borramos las 6 primeras letras: que tal estas
 Si cambiamos ESTAS por ESTAMOS: Hola, que tal estamos

Ejercicios propuestos:

(4.4.6.1) Una variante del ejercicio 4.4.5.2, que no distinga entre mayúsculas y minúsculas a la hora de buscar.

(4.4.6.2) Un programa que pida al usuario una frase y elimine todos los espacios redundantes que contenga (debe quedar sólo un espacio entre cada palabra y la siguiente).

4.4.7. Descomponer una cadena en fragmentos

Una operación relativamente frecuente, pero trabajosa, es descomponer una cadena en varios fragmentos que estén delimitados por ciertos separadores. Por ejemplo, podríamos descomponer una frase en varias palabras que estaban separadas por espacios en blanco.

Si lo queremos hacer "de forma artesanal", podemos recorrer la cadena buscando y contando los espacios (o los separadores que nos interesen). Así podremos saber el tamaño del array que deberá almacenar las palabras (por ejemplo, si hay dos espacios, tendremos tres palabras). En una segunda pasada, obtendremos las subcadenas que hay entre cada dos espacios y las guardaríamos en el array. No es especialmente sencillo.

Afortunadamente, C# nos permite hacerlo con **Split**, que crea un array a partir de los fragmentos de la cadena usando el separador que le indiquemos, así:

```
// Ejemplo_04_04_07a.cs
// Cadenas de texto: partir con "Split"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_04_07a
{
    public static void Main()
    {
        string ejemplo = "uno dos tres cuatro";
        char delimitador = ' ';
        int i;
```

```

        string [] ejemploPartido = ejemplo.Split(delimitador);

        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

```

Pero también podemos usar como delimitador un conjunto (un array de caracteres). Por ejemplo, podríamos considerar un separador válido el espacio, pero también la coma, el punto y cualquier otro. Los cambios en el programa serían mínimos:

```

// Ejemplo_04_04_07b.cs
// Cadenas de texto: partir con "Split" - 2
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_04_07b
{
    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]);
    }
}

```

Ejercicios propuestos:

(4.4.7.1) Un programa que pida al usuario una frase y muestre sus palabras en orden inverso.

(4.4.7.2) Un programa que pida al usuario varios números separados por espacios y muestre su suma.

4.4.8. 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, y eso 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("La frase es mayor que hola");
```

Esto tiene una limitación: si lo usamos de esa manera, las mayúsculas y minúsculas se consideran diferentes. Es más habitual que deseemos comparar sin distinguir entre mayúsculas y minúsculas, y eso se puede conseguir convirtiendo ambas cadenas a mayúsculas antes de convertir, o bien empleando String.Compare, al que indicamos las dos cadenas y un tercer dato booleano, que será "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_04_04_08a.cs
// Cadenas de texto y comparación alfabética
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_04_08a
{
    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)
```

Ejercicios propuestos:

(4.4.8.1) Un programa que pida al usuario dos frases y diga cual sería la "mayor" de ellas (la que aparecería en último lugar en un diccionario).

(4.4.8.2) Un programa que pida al usuario cinco frases, las guarde en un array y muestre la "mayor" de ellas.

4.4.9. Una cadena modificable: StringBuilder

Si tenemos la necesidad de modificar una cadena letra a letra, no podemos usar un "string" convencional, porque no es válido hacer cosas como `texto[1]='h'`; Deberíamos formar una nueva cadena en la que modificásemos esa letra a base de unir varios substring o de borrar un fragmento con `Remove` y añadir otro con `Insert`.

Como alternativa, podemos 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 ciertas ocasiones con los Arrays), y se pueden convertir a una cadena "convencional" usando "ToString":

```
// Ejemplo_04_04_09a.cs
// Cadenas modificables con "StringBuilder"
// Introducción a C#, por Nacho Cabanes

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

public class Ejemplo_04_04_09a
{
    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:

(4.4.9.1) Un programa que pida una cadena al usuario y la modifique, de modo que todas las vocales se conviertan en "o".

(4.4.9.2) Un programa que pida una cadena al usuario y la modifique, de modo que las letras de las posiciones impares (primera, tercera, etc.) estén en minúsculas y las de las posiciones pares estén en mayúsculas, mostrando el resultado en pantalla. Por ejemplo, a partir de un nombre como "Nacho", la cadena resultante sería "nAcHo".

(4.4.9.3) Crea un juego del ahorcado, en el que un primer usuario introduzca la palabra a adivinar, se muestre ésta 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.

4.5. Recorriendo arrays y cadenas con "foreach"

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

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

```
// Ejemplo_04_05a.cs
// Ejemplo de "foreach"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_05a
{
    public static void Main()
    {
        int[] diasMes = {31, 28, 21};
        foreach(int dias in diasMes) {
            Console.WriteLine("Dias 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();
    }
}
```

Ejercicios propuestos:

(4.5.1) Un programa que pida tu nombre y lo muestre con un espacio entre cada par de letras, usando "foreach".

(4.5.2) Un programa que pida al usuario una frase y la descomponga en subcadenas separadas por espacios, usando "Split". Luego debe mostrar cada subcadena en una línea nueva, usando "foreach".

(4.5.3) Un programa que pida al usuario varios números separados por espacios y muestre su suma (como el del ejercicio 4.4.7.2), pero empleando "foreach".

4.6. 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.

La única complicación real es que el array no estará completamente lleno: habrá espacio para 1000 datos, pero iremos añadiendo de uno en uno. Basta con contar el número de fichas que tenemos almacenadas, y así sabremos en qué posición iría la siguiente: 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 existentes, 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 ... */ }
```

(esta segunda alternativa es la que se suele considerar "más natural" para un programador en un lenguaje como C#).

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_04_06a.cs
// Tabla con muchos struct y menú para manejarla
// Introducción a C#, por Nacho Cabanes
```

```

using System;

public class Ejemplo_04_06a {

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

    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 preguntemos al usuario
        long tamanyoBuscar;    // Para buscar por tamaño

        do {
            // Menu principal, repetitivo
            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 por encima 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 ver?");
                    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
}
}

```

(Como quizá hayas notado, este fuente, que es un poco más largo que los anteriores, abre las llaves al final de cada línea -estilo Java- y usa tabulaciones de 2 espacios, en vez de 4, para ocupar menos espacio en papel y caber en el ancho de una página; recuerda que puedes usar estilo Java si lo prefieres, pero en general el fuente será más legible con tabulaciones de 4 espacios en vez de 2).

Este programa 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 descontado, 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:

(4.6.1) 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.

(4.6.2) Un programa que pida datos de 8 personas: nombre, día de nacimiento, mes de nacimiento, y año de nacimiento (que se deben almacenar en un array 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.

(4.6.3) 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").

(4.6.4) Mejorar la base de datos de ficheros (ejemplo 04_06a) para que no permita introducir tamaños incorrectos (números negativos) ni nombres de fichero vacíos.

(4.6.5) Ampliar la base de datos de ficheros (ejemplo 04_06a) 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 "Contains" o "IndexOf"). Esta búsqueda no debe distinguir mayúsculas y minúsculas (con la ayuda de ToUpper o ToLower).

(4.6.6) Ampliar el ejercicio anterior (4.6.5) para que la búsqueda sea incremental: el usuario irá indicando letra a letra el texto que quiere buscar, y se mostrará todos los datos que lo contienen (por ejemplo, primero los que contienen "j", luego "ju", después "jua" y finalmente "juan").

(4.6.7) Ampliar la base de datos de ficheros (ejemplo 04_06a) 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).

(4.6.8) Mejorar la base de datos de ficheros (ejemplo 04_06a) 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.

(4.6.9) Ampliar la base de datos de ficheros (ejemplo 04_06a) 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.

4.7. 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 único dato, y habrá que dar tantas pasadas como datos existen. Por tanto, para un array con 1.000 datos, podrían llegar a ser necesarias un millón de comparaciones (1.000 x 1.000).

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 comentaremos más adelante.

Veremos tres de estos métodos simples de ordenación, primero mirando la apariencia que tiene el algoritmo, y luego juntando los tres métodos 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_04_07a.cs
// Ordenaciones simples
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_07a
{
    public static void Main()
    {
        int[] datos = {5, 3, 14, 20, 8, 9, 1};
        int i,j,datoTemporal;
        int n=7; // Numero 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("Ordenando 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.WriteLine("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("Ordenando mediante inserción directa... ");

```

```

int[] datos3 = {5, 3, 14, 20, 8, 9, 1};
for(i=1; i < n; i++)
{
    foreach (int dato in datos3) // Muestro datos
        Console.WriteLine("{0} ",dato);

    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.WriteLine("{0} ",dato);
}
}

```

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

```

Ejercicios propuestos:

(4.7.1) Un programa que pida al usuario 6 números en coma flotante y los muestre ordenados de menor a mayor. Escoge el método de ordenación que prefieras.

(4.7.2) Un programa que pida al usuario 5 nombres y los muestre ordenados alfabéticamente (recuerda que para comparar cadenas no podrás usar el símbolo ">", sino "CompareTo").

(4.7.3) Un programa que pida al usuario varios números, los vaya añadiendo a un array, mantenga el array ordenado continuamente y muestre el resultado tras añadir cada nuevo dato (todos los datos se mostrarán en la misma línea, separados por espacios en blanco). Terminará cuando el usuario teclee "fin".

(4.7.4) Amplia el ejercicio anterior, para añadir una segunda fase en la que el usuario pueda "preguntar" si un cierto valor está en el array. Como el array está ordenado, la búsqueda no se hará hasta el final de los datos, sino hasta que se encuentre el dato buscado o un dato mayor que él.

Una vez que los datos están ordenados, podemos buscar uno concreto dentro de ellos empleando la **búsqueda binaria**: se comienza por el punto central; si el valor buscado es mayor que el del punto central, se busca esta vez sólo en la mitad superior (o en la inferior), y así sucesivamente, de modo que cada vez se busca entre un conjunto de datos que tiene la mitad de tamaño que el anterior. Esto puede suponer una enorme ganancia en velocidad: si tenemos 1.000 datos, una búsqueda lineal hará 500 comparaciones como media, mientras que una búsqueda lineal hará 10 comparaciones o menos. Se podría implementar así:

```
// Ejemplo_04_07b.cs
// Búsqueda binaria
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_07b
{
    public static void Main()
    {
        const int n = 1000;
        int[] datos = new int[n];
        int i,j,datoTemporal;

        // Primero generamos datos al azar
        Console.WriteLine("Generando... ");
        Random r = new Random();
        for(i=0; i < n; i++)
            datos[i] = r.Next(1, n*2);
```

```

// Luego los ordenamos mediante burbuja
Console.WriteLine("Ordenando... ");
for(i=0; i < n-1; i++)
{
    for(j=i+1; j < n; j++)
    {
        if (datos[i] > datos[j])
        {
            datoTemporal = datos[i];
            datos[i] = datos[j];
            datos[j] = datoTemporal;
        }
    }
}

// Mostramos los datos
foreach (int dato in datos)
    Console.WriteLine("{0} ",dato);
Console.WriteLine();

// Y comenzamos a buscar
int valorBuscado = 1001;
Console.WriteLine("Buscando si aparece {0}...", valorBuscado);

int limiteInferior = 0;
int limiteSuperior = 999;
bool terminado = false;

while(! terminado)
{
    int puntoMedio = limiteInferior+(limiteSuperior-limiteInferior) /
2;

    // Aviso de dónde buscamos
    Console.WriteLine("Buscando entre pos {0} y {1}, valores {2} y
{3}, "+
        " centro {4}:{5}",
        limiteInferior, limiteSuperior,
        datos[limiteInferior], datos[limiteSuperior],
        puntoMedio, datos[puntoMedio]);
    // Compruebo si hemos acertado
    if (datos[puntoMedio] == valorBuscado)
    {
        Console.WriteLine("Encontrado!");
        terminado = true;
    }
    // 0 si se ha terminado la búsqueda
    else if (limiteInferior == limiteSuperior-1)
    {
        Console.WriteLine("No encontrado");
        terminado = true;
    }
    // Si no hemos terminado, debemos seguir buscando en una mitad
    if ( datos[puntoMedio] < valorBuscado )
        limiteInferior = puntoMedio;
    else
        limiteSuperior = puntoMedio;
}
}

```

```
}
```

Ejercicios propuestos:

(4.7.5) Realiza una variante del ejercicio 4.7.4, que en vez de hacer una búsqueda lineal (desde el principio), use "búsqueda binaria": se comenzará a comparar con el punto medio del array; si nuestro dato es menor, se vuelve a probar en el punto medio de la mitad inferior del array, y así sucesivamente.

¿Y no se puede **ordenar de forma más sencilla**? Sí, existe un "Array.Sort" que hace todo por nosotros... pero recuerda que no (sólo) se trata de que conozcas la forma más corta posible de hacer un ejercicio, sino de que aprendas a resolver problemas...

```
// Ejemplo_04_07c.cs
// Array.Sort
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_04_07c
{
    public static void Main()
    {
        int[] datos = {5, 3, 14, 20, 8, 9, 1};
        Array.Sort(datos); // Ordeno

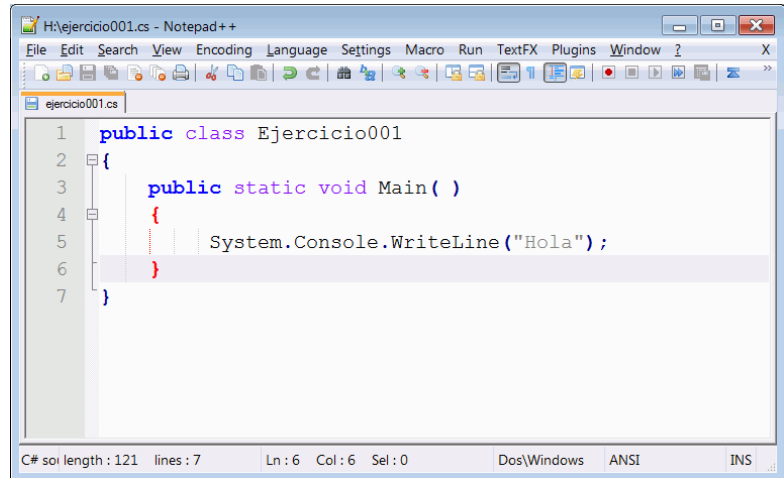
        foreach (int dato in datos) // Muestro datos finales
            Console.Write("{0} ", dato);
        Console.WriteLine();
    }
}
```

Ejercicios propuestos:

(4.7.6) Crea una variante del ejercicio 4.7.3, pero usando esta vez Array.Sort para ordenar: un programa que pida al usuario varios números, los vaya añadiendo a un array, mantenga el array ordenado continuamente y muestre el resultado tras añadir cada nuevo dato (todos los datos se mostrarán en la misma línea, separados por espacios en blanco). Terminará cuando el usuario teclee "fin".

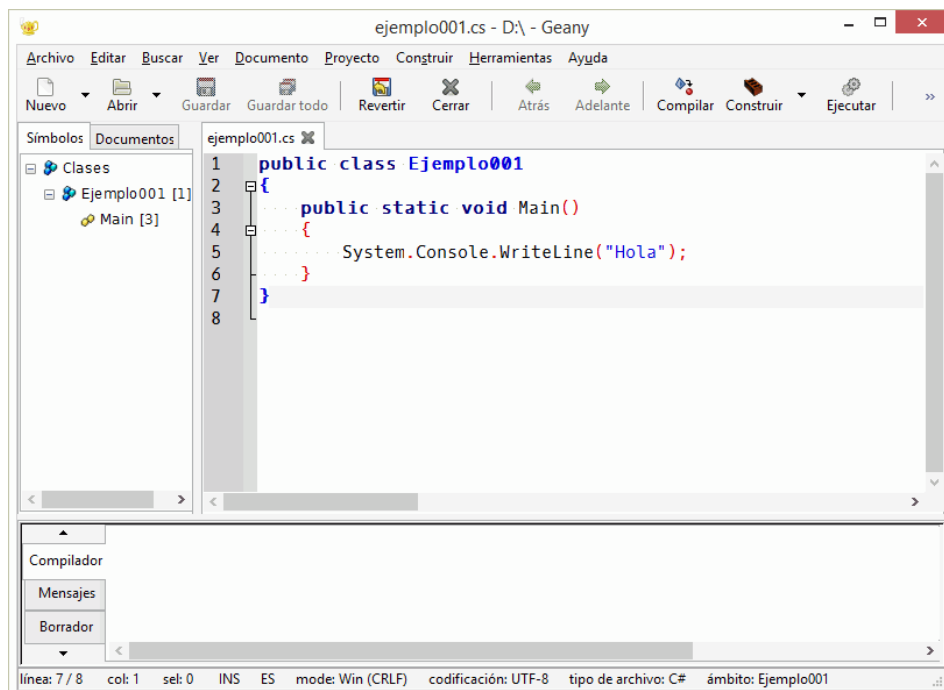
4.8. Otros editores más avanzados: Notepad++ y Geany

Ahora que conocemos los fundamentos, puede ser el momento de pasar a un editor de texto un poco más avanzado. Por ejemplo, en Windows podemos usar **Notepad++**, que es gratuito, destaca la sintaxis en colores, muestra el número de línea y columna en la que nos encontramos (lo que es útil para corregir errores), ayuda a encontrar las llaves emparejadas, realza la línea en la que nos encontramos, permite tabular y des-tabular bloques completos de código, tiene soporte para múltiples ventanas, etc.

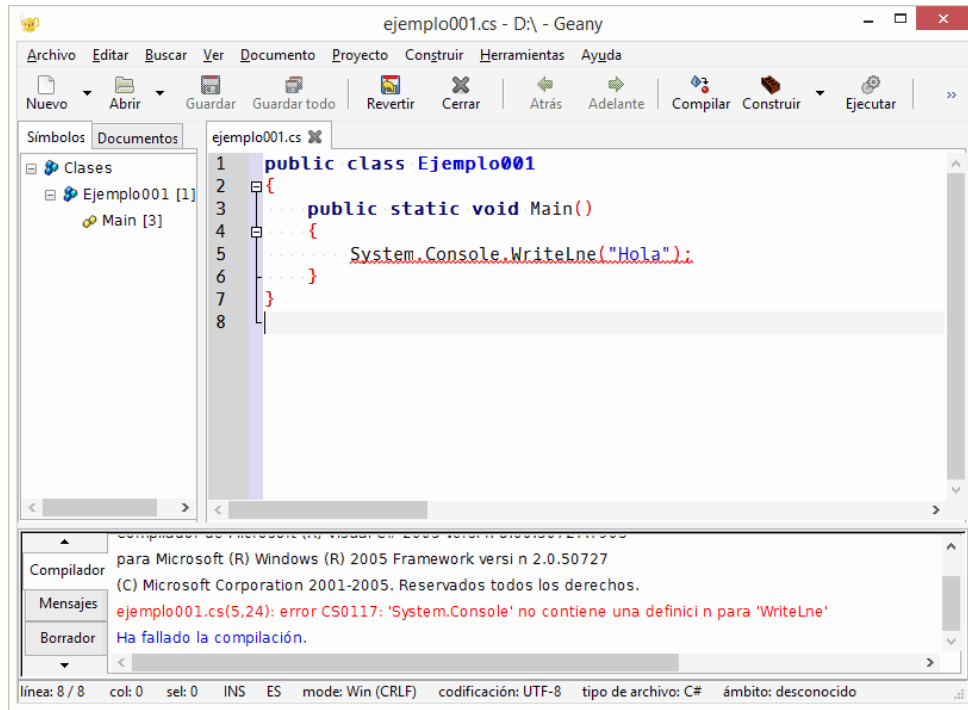


El problema de Notepad++ es que sigue siendo necesario teclear la orden de compilación, ya sea volviendo a la pantalla de consola o desde el menú Ejecutar (o Run, si tenemos una versión en inglés).

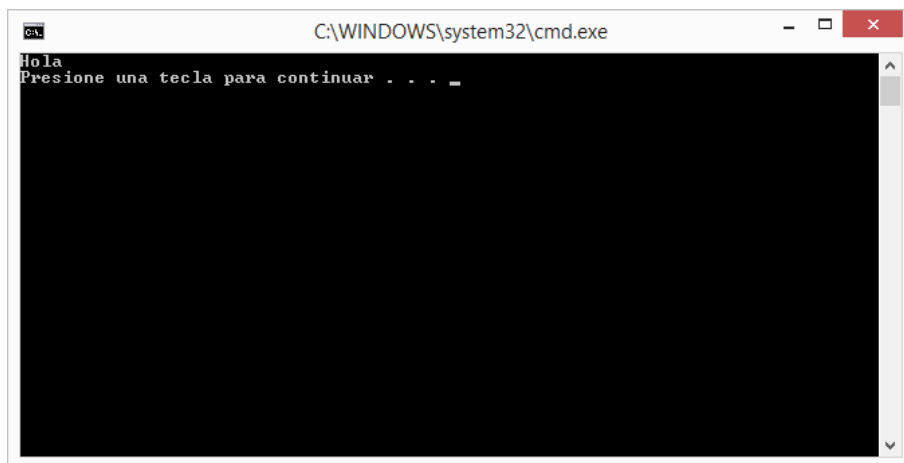
Por eso, un segundo editor que puede resultar aún más interesante (aunque a cambio tiene menos "plugins" para ampliar sus funcionalidades) es **Geany**, que sí permite compilar y ejecutar nuestros programas sin necesidad de salir del editor:



No es sólo que la barra de herramientas tenga botones para Compilar y para Ejecutar. Es que además, en caso de error de compilación, se nos mostrará la lista de errores y se destacará (con un subrayado rojo) las líneas causantes de esos errores:

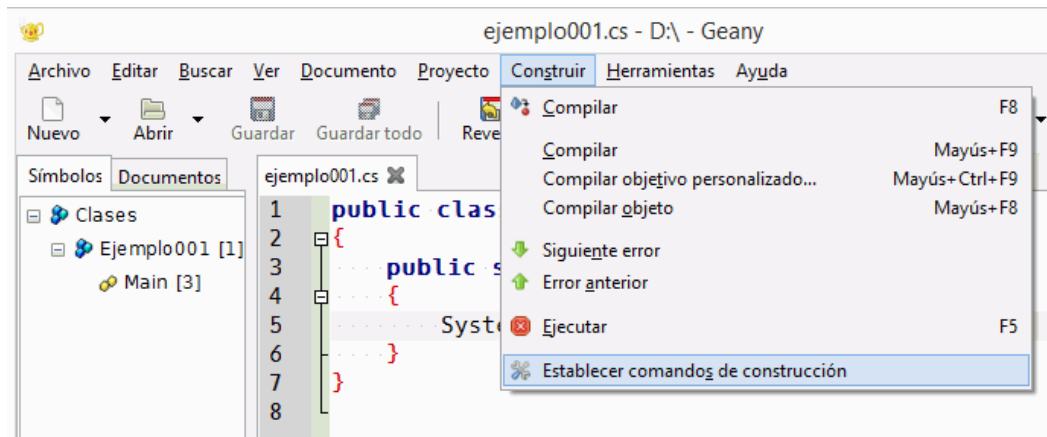


Si, por el contrario, todo ha sido correcto, la ventana que nos muestra la ejecución de nuestro programa se quedará pausada para que podamos comprobar los resultados:

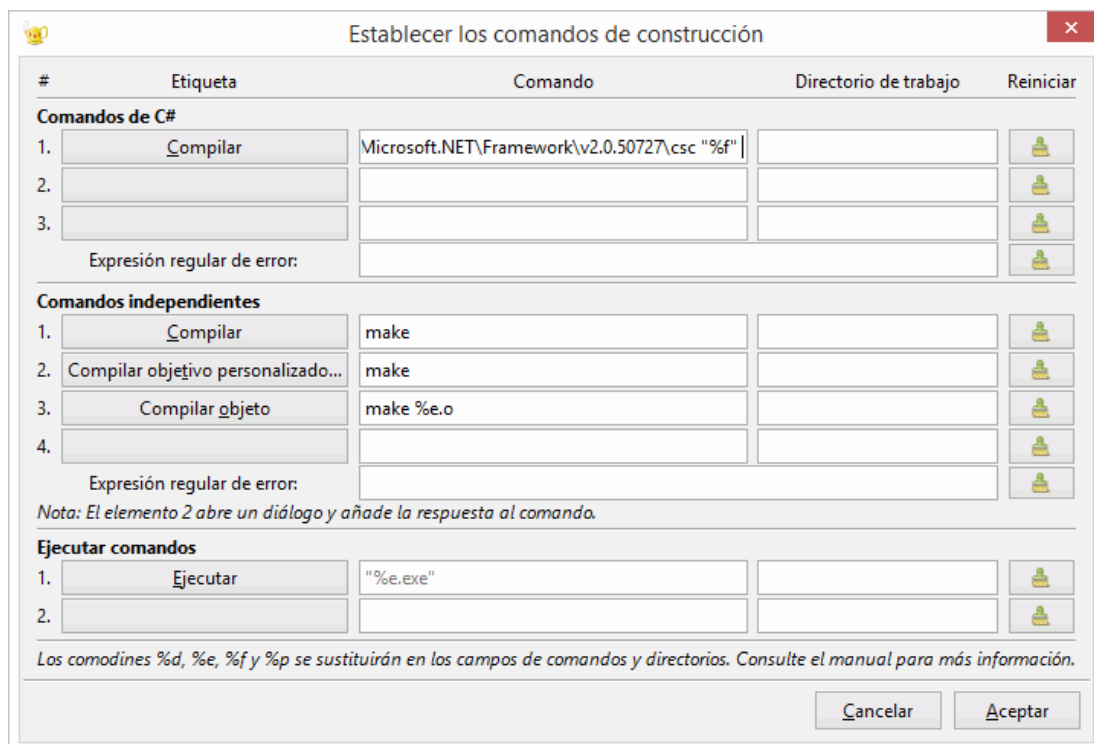


La instalación de Geany en Linux es trivial: basta descargarlo usando nuestro gestor de paquetes (Synaptic o el que sea) y todo suele funcionar a la primera (si habíamos instalado Mono previamente). En Windows es fácil descargarlo desde su página oficial (geany.org) pero no podremos compilar desde él directamente,

porque lo habitual es que no sepa en qué ruta se encuentra nuestro compilador. Lo podemos solucionar abriendo un fuente en lenguaje C# y entrando al menú "Construir", en la opción "Establecer comandos de construcción":



Nos aparecerá una ventana de la que nos interesan dos casillas: la que permite escribir el comando para compilar y la del comando para ejecutar.



Si queremos utilizar **Mono**: en la primera casilla, pondremos la ruta completa de nuestro compilador Mono (mcs.exe) seguida por "%f" para indicar que hay que detallar a Mono el nombre de nuestro fichero actual. En la casilla "Ejecutar", escribiremos la ruta de "mono.exe" seguida por "%e.exe", para que a Mono se le indique el nombre del ejecutable que habremos obtenido al compilar.

De forma alternativa, podemos usar el propio **compilador de ".Net"** que viene incluido con Windows, y que se encontrará en el disco de sistema (habitualmente C:), en la carpeta de Windows, subcarpeta "Microsoft.NET", dentro de "Framework" o "Framework64" (según si queremos usar la versión de 32 bits o de 64 bits), en una carpeta que indica el número de versión de la plataforma .Net (por ejemplo, "v2.0.50727" para la versión 2; puede haber varias versiones instaladas). El compilador se llama "csc.exe", y, al igual que para Mono, la línea debe terminar con "%f" para decir a Geany que cuando lance el compilador debe indicarle el nombre del fichero que estamos editando:

```
C:\Windows\Microsoft.NET\Framework\v2.0.50727\csc "%f"
```

Esto habría sido muy trabajoso para compilar desde línea de comando nuestros primeros programas, pero una vez que usamos editores más avanzados, sólo hay que configurarlo una vez en Geany y se utilizará automáticamente cada vez que creemos un fuente en C# desde ese editor.

En este caso, en la casilla "Ejecutar" bastará con escribir "%e.exe", para que se lance directamente el ejecutable recién creado:

```
"%e.exe"
```

Nota: Las comillas dobles que rodean a "%e.exe" (y a "%f") permiten que se lance correctamente incluso si hubiéramos escrito algún espacio en el nombre del fichero (costumbre poco recomendable para un fuente de un programa).

Por supuesto, existen **otros muchos editores gratuitos**, que puedes utilizar sin gastar dinero y que harán tu rutina de programador mucho más cómoda que con el Bloc de Notas. Van desde editores sencillos como Notepad2 o Scite hasta **entornos integrados**, que usaremos dentro de poco para programas de mayor tamaño, en los que intervendrán varios ficheros fuente de forma simultánea. Es el caso de Visual Studio, Sharp Develop y MonoDevelop (o su versión modernizada y ampliada, Xamarin Studio), que veremos un poco más adelante, cuando nuestros programas lleguen a ese nivel de complejidad.

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.
- Evitaremos mucho código repetitivo.
- 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 (aunque para proyectos realmente grandes, pronto veremos una alternativa que hace que el reparto y la integración sean más sencillos).

Esos "trozos" de programa son lo que suele llamar "subrutinas", "procedimientos" o "funciones". En el lenguaje C y sus derivados (entre los que está C#), 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.Write("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_05_02a.cs
// Funcion "Saludar"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_02a
{
    public static void Saludar()
    {
        Console.Write("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 o de una base de datos simple como las que hicimos en el tema anterior, podría ser simplemente:

```
LeerDatosDeFichero();
```

```
do {
    MostrarMenu();
    opcion = PedirOpcion();
    switch( opcion ) {
        case 1: BuscarDatos(); break;
        case 2: ModificarDatos(); break;
        case 3: AnadirDatos(); break;
        ...
    }
}
```

Ejercicios propuestos:

(5.2.1) Crea una función llamada "BorrarPantalla", que borre la pantalla dibujando 25 líneas en blanco. Crea también un "Main" que permita probarla.

(5.2.2) Crea una función llamada "DibujarCuadrado3x3", que dibuje un cuadrado formato por 3 filas con 3 asteriscos cada una. Incluye un "Main" para probarla.

(5.2.3) Descompón en funciones la base de datos de ficheros (ejemplo 04_06a), de modo que el "Main" sea breve y más legible (Pista: las variables que se compartan entre varias funciones deberán estar fuera de todas ellas, y deberán estar precedidas por la palabra "static").

5.3. *Parámetros de una función*

Es muy frecuente que nos interese indicarle a nuestra función ciertos datos con los que queremos que trabaje. Los llamaremos "parámetros" y los indicaremos dentro del paréntesis que sigue al nombre de la función, separados por comas. Para cada uno de ellos, deberemos indicar su tipo de datos (por ejemplo "int") y luego su nombre.

Por ejemplo, si escribimos en pantalla números reales con frecuencia, nos puede resultar útil crear una función auxiliar que nos los muestre con el formato que nos interese (que podría ser con exactamente 3 decimales). Lo podríamos hacer así:

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

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

```
EscribirNumeroReal(2.3f);
```

(recordemos que el sufijo "f" sirve 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 pasando un dato "double" a una función que espera un "float").

El programa completo podría quedar así:

```
// Ejemplo_05_03a.cs
// Funcion "EscribirNumeroReal"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_03a
{
    public static void EscribirNumeroReal( float n )
    {
        Console.WriteLine( n.ToString("#.###") );
    }

    public static void Main()
    {
        float x;

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

Como ya hemos anticipado, si hay más de un parámetro, deberemos indicar el tipo y el nombre para cada uno de ellos (incluso si todos son del mismo tipo), y separarlos entre comas:

```
public static void EscribirSuma( int a, int b )
{
    ...
}
```

De modo que un programa completo de ejemplo para una función con dos parámetros podría ser:

```
// Ejemplo_05_03b.cs
// Funcion "EscribirSuma"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_03b
{
    public static void EscribirSuma( int a, int b )
```

```

{
    Console.Write( a+b );
}

public static void Main()
{
    Console.Write("La suma de 4 y 7 es: ");
    EscribirSuma(4, 7);
}
}

```

Como se ve en estos ejemplos, se suele seguir un par de **convenios**:

- Ya que las funciones expresan acciones, en general su nombre será un verbo.
- En C# se recomienda que los elementos públicos se escriban comenzando por una letra mayúscula (y recordemos que, hasta que conozcamos las alternativas y el motivo para usarlas, nuestras funciones comienzan con la palabra "public"). Este criterio depende del lenguaje. Por ejemplo, en lenguaje Java es habitual seguir el convenio de que los nombres de las funciones deban comenzar con una letra minúscula.

Ejercicios propuestos:

(5.3.1) Crea una función "DibujarCuadrado" que dibuje en pantalla un cuadrado del ancho (y alto) que se indique como parámetro. Completa el programa con un Main que permita probarla.

(5.3.2) Crea una función "DibujarRectangulo" que dibuje en pantalla un rectángulo del ancho y alto que se indiquen como parámetros. Incluye un Main para probarla.

(5.3.3) Crea una función "DibujarRectanguloHueco" que dibuje en pantalla un rectángulo hueco del ancho y alto que se indiquen como parámetros, formado por una letra que también se indique como parámetro. Completa el programa con un Main que pida esos datos al usuario y dibuje el rectángulo.

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, que es el resultado de una

operación (por ejemplo, la raíz cuadrada de un número tiene como resultado otro número).

De igual modo, para nosotros también será habitual crear funciones que realicen una serie de cálculos y nos "devuelvan" (**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;
}
```

En este caso, nuestra función no es "void", sino "int", porque va a **devolver un número entero**. Eso sí, todas nuestras funciones seguirán siendo "public static" hasta que avancemos un poco más.

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

```
resultado = Cuadrado( 5 );
```

En general, en las operaciones matemáticas, no será necesario que el nombre de la función sea un verbo. El programa debería ser suficientemente legible si el nombre expresa qué operación se va a realizar en la función.

Un programa más detallado de ejemplo podría ser:

```
// Ejemplo_05_04a.cs
// Funcion "Cuadrado"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_04a
{
    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));
}
}

```

Podremos devolver cualquier tipo de datos, no sólo números enteros. Como segundo ejemplo, podemos hacer una función que nos diga cuál es el mayor de dos números reales así:

```

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

```

Como se ve en este ejemplo, una función puede tener más de un "return".

En cuanto se alcance un "return", se sale de la función por completo. Eso puede hacer que una función mal diseñada haga que el compilador nos dé un aviso de "código inalcanzable", como en el siguiente ejemplo:

```

public static string Inalcanzable()
{
    return "Aquí sí llegamos";

    string ejemplo = "Aquí no llegamos";
    return ejemplo;
}

```

Ejercicios propuestos:

(5.4.1) Crea una función "Cubo" que calcule el cubo de un número real (float) que se indique como parámetro. El resultado deberá ser otro número real. Prueba esta función para calcular el cubo de 3.2 y el de 5.

(5.4.2) Crea una función "Menor" que calcule el menor de dos números enteros que recibirá como parámetros. El resultado será otro número entero.

(5.4.3) Crea 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.

(5.4.4) Crea una función "Inicial", que devuelva la primera letra de una cadena de texto. Prueba esta función para calcular la primera letra de la frase "Hola".

(5.4.5) Crea una función "UltimaLetra", que devuelva la última letra de una cadena de texto. Prueba esta función para calcular la última letra de la frase "Hola".

(5.4.6) Crea una función "MostrarPerimSuperfCuadrado" que reciba un número entero y calcule y muestre en pantalla el valor del perímetro y de 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 las variables se comportarán de forma distinta según donde las declaremos.

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. Por ahora, para nosotros, una variable global deberá llevar siempre la palabra "**static**" (dentro de poco veremos el motivo real y cuándo no será necesario).

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 no es a través de variables globales, sino usando los parámetros de cada función y los valores devueltos, como en el anterior ejemplo. Aun así, esta restricción es menos grave en lenguajes modernos, como C#, que en otros lenguajes más antiguos, como C, porque, como veremos en el próximo tema, el hecho de descomponer un programa en varias clases minimiza los efectos negativos de esas variables que se comparten entre varias funciones, además de que muchas veces tendremos datos compartidos, que no serán realmente "variables globales" sino datos específicos del problema, que llamaremos "**atributos**".

Vamos a ver el uso de variables locales 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_05_05a.cs
// Ejemplo de función con variables locales
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_05a
{
    public static int Potencia(int nBase, int nExponente)
    {
        int temporal = 1;          // Valor inicial que voy incrementando

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

        return temporal; // Al final, 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". Este ejemplo no contiene ninguna variable global.

(Nota: el parámetro no se llama "base" sino "nBase" porque la palabra "base" es una palabra reservada en C#, que no podremos usar como nombre de variable).

Ejercicios propuestos:

(5.5.1) Crea 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, volvérselo a pedir en caso de error, y devolver un valor correcto. Pruébalo con un programa que pida al usuario un año entre 1800 y 2100.

(5.5.2) Crea 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 "3x0=0" hasta "3x10=30").

(5.5.3) Crea una función "EsPrimo", que reciba un número y devuelva el valor booleano "true" si es un número primo o "false" en caso contrario.

(5.5.4) Crea una función "ContarLetra", 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 (porque la "a" aparece 2 veces).

(5.5.5) Crea una función "SumaCifras" que reciba un número cualquiera y que devuelva como resultado la suma de sus dígitos. Por ejemplo, si el número fuera 123 la suma sería 6.

(5.5.6) Crea una función "Triángulo" 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 distintas? Vamos a comprobarlo con un ejemplo:

```
// Ejemplo_05_06a.cs
// Dos variables locales con el mismo nombre
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_05_06a
{
```

```

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 variables tengan el mismo nombre no afecta al funcionamiento del programa, siguen siendo distintas, porque cada una está en un bloque ("ámbito") distinto.

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

```

// Ejemplo_05_06b.cs
// Una variable global
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_06b
{
    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);
    }
}

```

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_05_07a.cs
// Modificar una variable recibida como parámetro - acercamiento
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_07a
{
    public static void Duplicar(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);
        Duplicar(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, **no se conservan los cambios** que hagamos a esa variable que se ha recibido como parámetro.

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 las modificaciones 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_05_07b.cs
// Modificar una variable recibida como parámetro - correcto
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_07b
{
    public static void Duplicar(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);
        Duplicar(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**". No podemos llamar a una función que tenga parámetros por referencia si los parámetros no

tienen valor inicial. Por ejemplo, una función que devuelva la primera y segunda letra de una frase sería así:

```
// Ejemplo_05_07c.cs
// Parámetros "out"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_07c
{
    public static void DosPrimerasLetras(string cadena,
        out char l1, out char l2)
    {
        l1 = cadena[0];
        l2 = cadena[1];
    }

    public static void Main()
    {
        char letra1, letra2;
        DosPrimerasLetras("Nacho", out letra1, out letra2);
        Console.WriteLine("Las dos primeras letras son {0} y {1}",
            letra1, letra2);
    }
}
```

Si pruebas este ejemplo, verás que no compila si cambias "out" por "ref", a no ser que des valores iniciales a "letra1" y "letra2".

Ejercicios propuestos:

(5.7.1) Crea una función "Intercambiar", que intercambie el valor de los dos números enteros que se le indiquen como parámetro. Crea también un programa que la pruebe.

(5.7.2) Crea 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. Crea un "Main" que te permita comprobar que funciona correctamente.

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 describir el ejemplo 05_07b, de modo que "Main" aparezca en primer lugar y "Duplicar" aparezca después, y seguiría compilando y funcionando igual:

```
// Ejemplo_05_08a.cs
// Función tras Main
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_08a
{
    public static void Main()
    {
        int n = 5;
        Console.WriteLine("n vale {0}", n);
        Duplicar(ref n);
        Console.WriteLine("Ahora n vale {0}", n);
    }

    public static void Duplicar(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" (una única vez), y luego llamaremos a "Next" cada vez que queramos obtener valores entre dos extremos:

```
// Creamos un objeto Random
Random generador = new Random();

// Generamos un número entre dos valores dados
// (el segundo límite no está incluido)
int aleatorio = generador.Next(1, 101);
```

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

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

Pero esta forma simplificada no sirve si necesitamos obtener dos números aleatorios a la vez, porque los dos se obtendrían en el mismo milisegundo y tendrían el mismo valor; en ese caso, no habría más remedio que utilizar "Random" y llamar dos veces a "Next".

Vamos a ver un ejemplo, que muestre en pantalla dos números al azar:

```
// Ejemplo_05_09_01a.cs
// Números al azar
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_09_01a
{
    public static void Main()
    {
        Random r = new Random();
        int aleatorio = r.Next(1, 11);
        Console.WriteLine("Un número entre 1 y 10: {0}", aleatorio);
        int aleatorio2 = r.Next(10, 21);
        Console.WriteLine("Otro entre 10 y 20: {0}", aleatorio2);
    }
}
```

Ejercicios propuestos:

(5.9.1.1) Crea un programa que imite el lanzamiento de un dado, generando un número al azar entre 1 y 6.

(5.9.1.2) Crea un programa que genere un número al azar entre 1 y 100. El usuario tendrá 6 oportunidades para acertarlo.

(5.9.1.3) Mejora el programa del ahorcado (4.4.9.3), para que la palabra a adivinar no sea tecleada por un segundo usuario, sino que se escoja al azar de un "array" de palabras prefijadas (por ejemplo, nombres de ciudades).

(5.9.1.4) Crea un programa que genere un array relleno con 100 números reales al azar entre -1000 y 1000. Luego deberá calcular y mostrar su media.

(5.9.1.5) Crea un programa que "dibuje" asteriscos en 100 posiciones al azar de la pantalla. Para ayudarte para escribir en cualquier coordenada, puedes usar un array de dos dimensiones (con tamaños 24 para el alto y 79 para el ancho), que primero rellenes y luego dibujes en pantalla.

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"; en el caso de las funciones trigonométricas, el ángulo se debe indicar en radianes, no en grados)

y también tenemos 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, que casi cualquier programador pueda necesitar:

- 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);`

Un ejemplo más avanzado, usando funciones trigonométricas, que calculase el "coseno de 45 grados" podría ser:

```
// Ejemplo_05_09_02a.cs
// Ejemplo de funciones trigonometricas
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_09_02a
{
    public static void Main()
    {
        double anguloGrados = 45;
        double anguloRadianes = anguloGrados * Math.PI / 180.0;

        Console.WriteLine("El coseno de 45 grados es: {0}",
            Math.Cos(anguloRadianes));
    }
}
```

Ejercicios propuestos:

(5.9.2.1) Crea un programa que halle (y muestre) la raíz cuadrada del número que introduzca el usuario. Se repetirá hasta que introduzca 0.

(5.9.2.2) Crea un programa que halle cualquier raíz (de cualquier orden) 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.

(5.9.2.3) Haz 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 "CalcularRaicesSegundoGrado", que recibirá como parámetros los coeficientes a, b y c (por valor), así como las soluciones x1 y x2 (por referencia). Deberá devolver los valores de las dos soluciones x1 y x2. Si alguna solución no existe, se devolverá como valor 100.000 para esa solución. Pista: la solución se calcula con

$$x = -b \pm \text{raíz}(b^2 - 4 \cdot a \cdot c) / (2 \cdot a)$$

(5.9.2.4) Haz un programa que pida al usuario 5 datos numéricos enteros, los guarde en un array, pida un nuevo dato y muestre el valor del array que se encuentra más cerca de ese dato, siendo mayor que él, o el texto "Ninguno es mayor" si ninguno lo es.

(5.9.2.5) Crea un programa que pida al usuario 5 datos numéricos reales, los guarde en un array, pida un nuevo dato y muestre el valor del array que se encuentra más cerca de ese dato en valor absoluto (es decir, el más próximo, sea mayor que él o menor que él).

(5.9.2.6) Crea una función "Distancia", que calcule la distancia entre dos puntos (x_1, y_1) y (x_2, y_2) , usando la expresión $d = \text{raíz}[(x_1 - x_2)^2 + (y_1 - y_2)^2]$.

(5.9.2.7) Crea un programa que pida al usuario un ángulo (en grados) y muestre su seno, coseno y tangente. Recuerda que las funciones trigonométricas esperan que el ángulo se indique en radianes, no en grados. La equivalencia es que 360 grados son 2π radianes.

(5.9.2.8) Crea un programa que muestre los valores de la función $y = 10 * \text{seno}(x/5)$, para valores de x entre 0 y 72 grados.

(5.9.2.9) Crea un programa que "dibuje" la gráfica de la función $y = 10 * \text{seno}(x/5)$, para valores de x entre 0 y 72 grados. Para ayudarte para escribir en cualquier coordenada, puedes usar un array de dos dimensiones, que primero rellenes y luego dibujes en pantalla (mira el ejercicio 5.9.1.5).

(5.9.2.10) Crea un programa que "dibuje" un círculo dentro de un array de dos dimensiones, usando las ecuaciones $x = x_{\text{Centro}} + \text{radio} * \cos(\text{ángulo})$, $y = y_{\text{Centro}} + \text{radio} * \sin(\text{ángulo})$. Si tu array es de 24x79, las coordenadas del centro serían (12,40). Recuerda que el ángulo se debe indicar en radianes (mira el ejercicio 5.9.1.5 y el 5.9.2.9).

5.9.3. Pero hay muchas más funciones...

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 (que devuelve la cadena que se ha introducido por teclado) o Console.WriteLine (que es "void", no devuelve nada). Nos iremos encontrando con otras muchas funciones a medida que avancemos.

5.10. Recursividad

La recursividad consiste en resolver un problema a partir de casos más simples del mismo problema. Una función recursiva es aquella que se "llama a ella misma", reduciendo la complejidad paso a paso hasta llegar a un caso trivial.

Dentro de las matemáticas tenemos varios ejemplos de funciones recursivas. Uno clásico es el "factorial de un número":

El factorial de 1 es 1:

$$1! = 1$$

Y el factorial de un número arbitrario es el producto de ese número por los que le siguen, hasta llegar a uno:

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

(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 se podría programar así:

```
// Ejemplo_05_10a.cs
// Funciones recursivas: factorial
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_10a
{
    public static long Factorial(int n)
    {
        if (n==1)                // Aseguramos que termine (caso base)
            return 1;
        return n * Factorial(n-1); // Si no es 1, sigue la recursión
    }

    public static void Main()
    {
        int num;
        Console.WriteLine("Introduzca un número entero: ");
        num = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Su factorial es: {0}", Factorial(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". Debemos encontrar un "caso trivial" que alcanzar, y un modo de disminuir la complejidad del problema acercándolo a ese caso.

- 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? 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. Los ejercicios propuestos te ayudarán a descubrir otros ejemplos de situaciones en las que se puede aplicar la recursividad.

Ejercicios propuestos:

(5.10.1) Crea 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. Piensa cuál será el caso base (qué potencia se puede calcular de forma trivial) y cómo pasar del caso "n-1" al caso "n" (por ejemplo, si sabes el valor de 5^4 , cómo hallarías el de 5^5 a partir de él).

(5.10.2) Como alternativa, crea 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".

(5.10.3) Crea 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).

(5.10.4) Crea un programa que emplee recursividad para calcular la suma de los elementos de un vector de números enteros, desde su posición inicial a la final, usando una función recursiva que tendrá la apariencia: SumaVector(v, desde, hasta). Nuevamente, piensa cuál será el caso base (cuántos elementos podrías sumar para que dicha suma sea trivial) y cómo pasar del caso "n-1" al caso "n" (por ejemplo, si conoces la suma de los 6 primeros elementos y el valor del séptimo elemento, cómo podrías emplear esta información para conocer la suma de los 7 primeros).

(5.10.5) Crea un programa que emplee recursividad para calcular el mayor de los elementos de un vector. El planteamiento será muy similar al del ejercicio anterior.

(5.10.6) Crea un programa que emplee recursividad para dar la vuelta a una cadena de caracteres (por ejemplo, a partir de "Hola" devolvería "aloH"). La función recursiva se llamará "Invertir(cadena)". Como siempre, analiza cuál será el caso base (qué longitud debería tener una cadena para que sea trivial darle la vuelta) y cómo pasar del caso "n-1" al caso "n" (por ejemplo, si ya has invertido las 5 primeras letras, que ocurriría con la de la sexta posición).

(5.10.7) Crea, 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.

(5.10.8) 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 $\text{MCD}(n, r)$.

(5.10.9) Crea dos funciones que sirvan para saber si un cierto texto es subcadena de una cadena. No puedes usar "Contains" ni "IndexOf", sino que debes analizar letra a letra. Una función debe ser iterativa y la otra debe ser recursiva.

(5.10.10) Crea una función que reciba una cadena de texto, y una subcadena, y devuelva cuántas veces aparece la subcadena en la cadena, como subsecuencia formada a partir de sus letras en orden. Por ejemplo, si recibes la palabra "Hhoola" y la subcadena "hola", la respuesta sería 4, porque se podría tomar la primera H con la primera O (y con la L y con la A), la primera H con la segunda O, la segunda H con la primera O, o bien la segunda H con la segunda O. Si recibes "hobla", la respuesta sería 1. Si recibes "ohla", la respuesta sería 0, porque tras la H no hay ninguna O que permita completar la secuencia en orden.

(5.10.11) El algoritmo de ordenación conocido como "Quicksort", parte de la siguiente idea: para ordenar un array entre dos posiciones "i" y "j", se comienza por tomar un elemento del array, llamado "pivote" (por ejemplo, el punto medio); luego se recoloca el array de modo que los elementos menores que el pivote queden a su izquierda y los mayores a su derecha; finalmente, se llama de forma recursiva a Quicksort para cada una de las dos mitades. El caso base de la función recursiva es cuando se llega a un array de tamaño 0 ó 1. Implementa una función que ordene un array usando este método.

5.11. Parámetros y valor de retorno de "Main"

Es muy frecuente que un programa llamado 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 .cs haciendo

```
ls -l *.cs
```

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

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

```
dir *.cs
```

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

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 puede leer desde un fichero BAT o CMD usando "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 (igual que antes, por convenio, devolviendo 0 si todo ha funcionado correctamente u otro código en caso contrario):

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

```
{
    ...
    return 0;
}
```

Un ejemplo que pusiera todo esto a prueba podría ser:

```
// Ejemplo_05_11a.cs
// Parámetros y valor de retorno de "Main"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_05_11a
{
    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("No ha indicado ningún parámetro!");
            Environment.Exit(1);
        }

        return 0;
    }
}
```

Ejercicios propuestos:

(5.11.1) Crea un programa llamado "suma", que calcule (y muestre) la suma de dos números que se le indiquen como parámetros en línea de comandos. Por ejemplo, si se teclea "suma 2 3" deberá responder "5", si se teclea "suma 2" responderá "2" y si se teclea únicamente "suma" deberá responder "no hay suficientes datos" y devolver un código de error 1.

(5.11.2) Crea 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".

(5.11.3) Crea una variante del ejercicio 5.11.2, en la que Main devuelva el código 1 si la operación indicada no es válida o 0 cuando sí sea una operación aceptable.

(5.11.4) Crea una variante del ejercicio 5.11.3, en la que Main devuelva también el código 2 si alguno de los dos números con los que se quiere operar no tiene un valor numérico válido.

6. Programación orientada a objetos

6.1. *¿Por qué los objetos?*

Cuando tenemos que realizar un proyecto grande, será necesario descomponerlo en varios subprogramas, de forma que podamos repartir el trabajo entre varias personas.

Esta descomposición no debe ser arbitraria. Por ejemplo, será deseable que cada bloque tenga unas responsabilidades claras, y que cada bloque no dependa de los detalles internos de otros bloques.

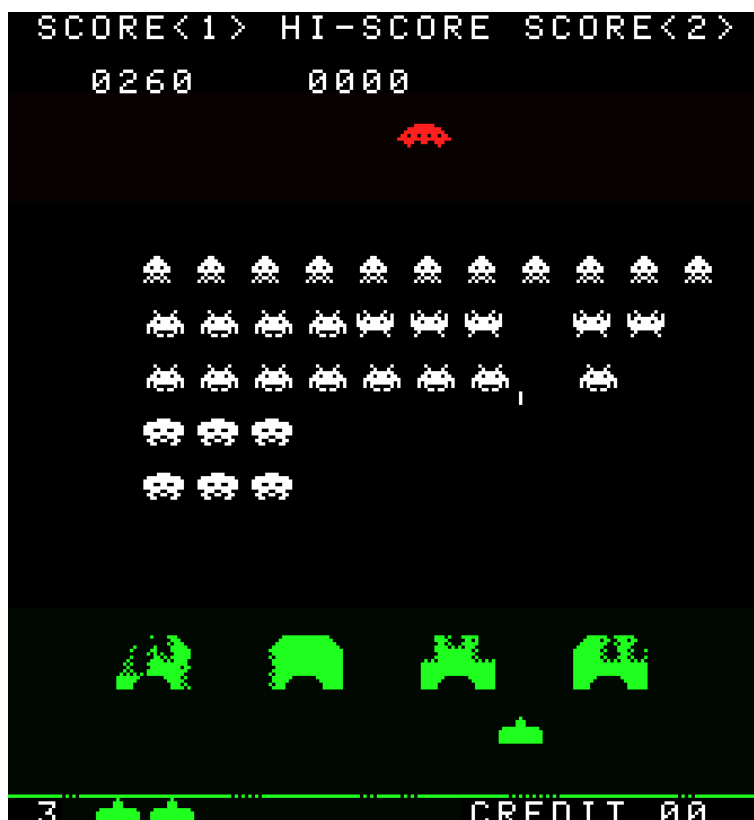
Existen varias formas de descomponer un proyecto, pero posiblemente la más recomendable consiste en tratar de verlo como una serie de "objetos" que colaboran entre sí.

Una forma de "descubrir" los objetos que forman parte de un programa es partir de la descripción del problema, y subrayar los nombres con un color y los verbos con otro color.

Como ejemplo, vamos a dedicar un momento a pensar qué elementos ("objetos") hay en un juego como el clásico **Space Invaders**. Cuando entramos al juego, aparece una pantalla de bienvenida, que nos recuerda detalles como la cantidad de puntos que obtendremos al "destruir" cada "enemigo":



Y cuando comenzamos una partida, veremos una pantalla como ésta, que vamos a analizar con más detalle:



Observando la pantalla anterior, o (preferiblemente) tras jugar algunas partidas, podríamos hacer una primera descripción del juego en lenguaje natural (que posiblemente habría que refinar más adelante, pero que nos servirá como punto de partida):

Nosotros manejamos una "**nave**", que se puede **mover** a izquierda y derecha y que puede **disparar**. Nuestra nave se esconde detrás de "**torres defensivas**", que se **destruyen** poco a poco cuando les impactan los **disparos**. Nos atacan (nos **disparan**) "**enemigos**". Además, estos enemigos se **mueven** de lado a lado, pero no de forma independiente, sino como un "**bloque**". En concreto, hay tres "tipos" de enemigos, que no se diferencian en su comportamiento, pero sí en su imagen. Tanto nuestro disparo como los de los enemigos **desaparecen** cuando salen de la pantalla o cuando impactan con algo. Si un disparo del enemigo impacta con nosotros, **perderemos una vida**; si un disparo nuestro impacta con un enemigo, lo **destruye**. Además, en ocasiones aparece un "**OVNI**" en la parte superior de la pantalla, en la parte superior de la pantalla, que se **mueve** del lado izquierdo al lado derecho y nos permite obtener puntuación extra si le impactamos con un disparo. Igualmente, hay un "**marcador**", que **muestra** la puntuación actual (que se irá **incrementando**) y el récord (mejor puntuación hasta el momento). El marcador se **reinicia** al comienzo de cada partida. Antes de cada "**partida**", pasamos por una pantalla de "**bienvenida**", que muestra una animación que nos informa de cuántos puntos obtenemos al destruir cada tipo de enemigo.

A partir de esa descripción, podemos buscar los **nombres** (puede ayudar si los subrayamos con un rotulador de marcar), que indicarán los objetos en los que podemos descomponer el problema, y los **verbos** (con otro rotulador de marcar, en otro color), que indicarán las acciones que puede realizar cada uno de esos objetos.

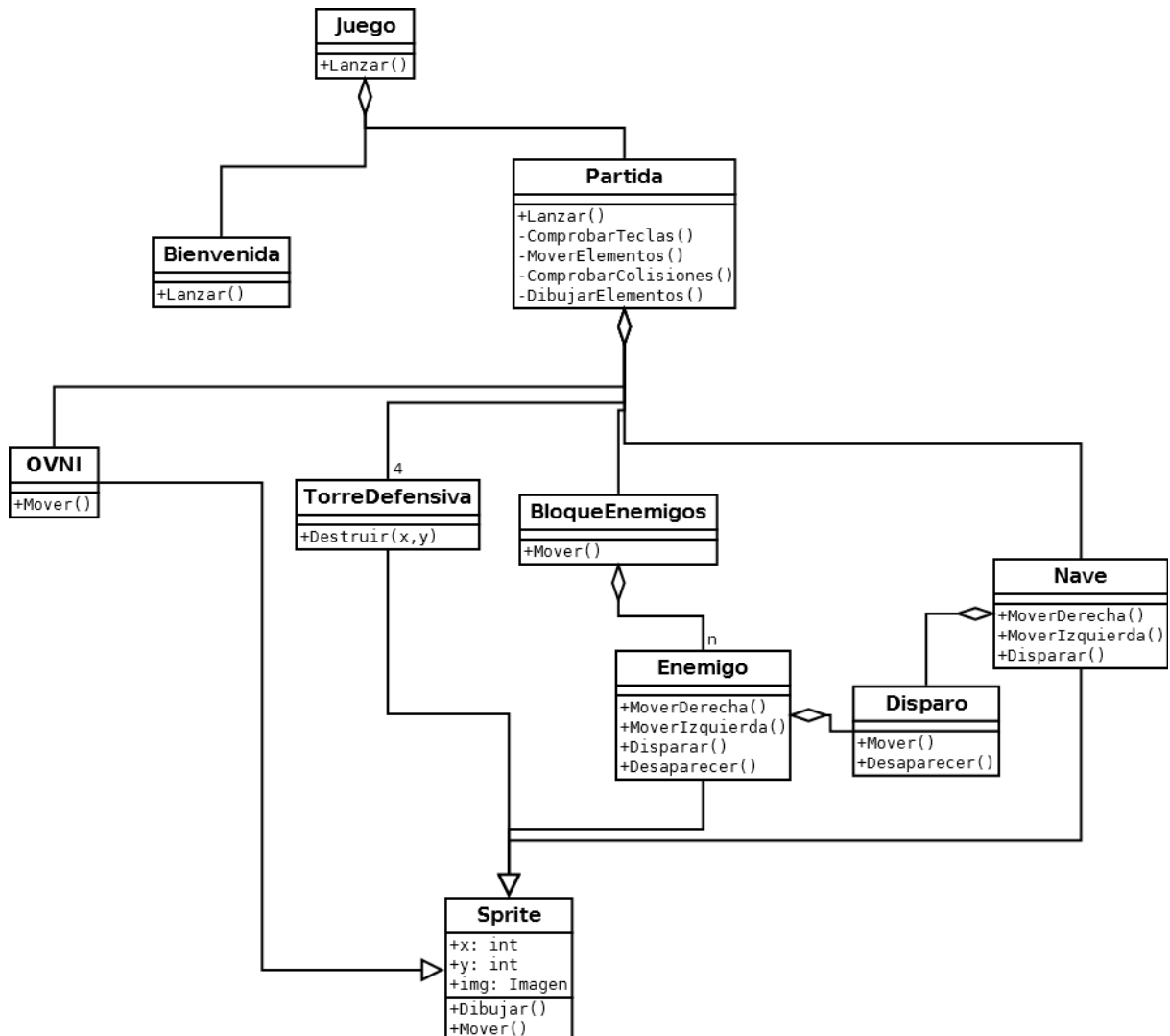
De la descripción subrayada de este juego concreto podemos extraer los siguientes objetos y las siguientes acciones:

- Nave : mover izquierda, mover derecha, disparar, perder vida
- Torre defensiva: destruir (un fragmento, en ciertas coordenadas)
- Enemigos : mover, disparar, desaparecer
- Ovni : mover
- Bloque (formado por enemigos) : mover
- Disparo : mover, desaparecer
- Marcador : mostrar, reiniciar, incrementar puntos
- Partida (contiene nave, enemigos, torres, ovni)

- Juego (formado por bienvenida y partida)

(En general, esta descomposición no tiene por qué ser única, distintos programadores o analistas pueden llegar a soluciones parcialmente distintas).

Esa serie de objetos, con sus relaciones y sus acciones, se puede expresar mediante un "**diagramas de clases**", que en nuestro caso podría ser así (simplificado):



El nombre "diagrama de clases" se debe a que se llama "**clase**" a un conjunto de objetos que tienen una serie de características comunes. Por ejemplo, un Honda Civic Type-R con matrícula 0001-AAA sería un objeto concreto perteneciente a la clase "Coche".

Algunos de los detalles que se pueden leer de ese diagrama (y que deberían parecerse bastante a la descripción inicial) son:

- El **objeto** principal de nuestro proyecto se llama "Juego" (el diagrama típicamente se leerá de arriba a abajo).
- El juego contiene una "Bienvenida" y una "Partida" (ese relación de que un objeto **"contiene"** a otros se indica mediante un rombo en el extremo de la línea que une ambas clases, junto a la clase "contenedora").
- La "Bienvenida" se puede "Lanzar" al comienzo del juego.
- Si así lo elige el jugador, se puede "Lanzar" una "Partida".
- En una partida participan una "Nave", cuatro "Torres" defensivas, un "BloqueDeEnemigos" formado por varios "Enemigos" (que, a su vez, podrían ser de tres tipos distintos, pero no afinaremos tanto por ahora) y un "Ovni".
- El "Ovni" se puede "Mover".
- Una "TorreDefensiva" se puede "Destruir" poco a poco, a partir de un impacto en ciertas coordenadas x,y.
- El "BloqueDeEnemigos" se puede "Mover".
- Cada "Enemigo" individual se puede mover a la derecha, a la izquierda, puede disparar o puede desaparecer (cuando un disparo le acierte).
- El "Disparo" se puede "Mover" y puede "Desaparecer".
- Nuestra "Nave" se puede mover a la derecha, a la izquierda y puede disparar.
- Tanto la "Nave" como las "Torres", los "Enemigos" y el "Ovni" son tipos concretos de "Sprite" (esa **relación entre un objeto más genérico y uno más específico** se indica con las puntas de flecha, que señalan al objeto más genérico).
- Un "Sprite" es una figura gráfica de las que aparecen en el juego. Cada sprite tendrá detalles (que llamaremos **"atributos"**) como una "imagen" y una posición, dada por sus coordenadas "x" e "y". Será capaz de hacer operaciones (que llamaremos **"métodos"**) como "dibujarse" (aparecer en pantalla) o "moverse" a una nueva posición. Cuando toda esta estructura de clases se convierte en un programa, los atributos se representarán **variables**, mientras que los "métodos" serán **funciones**. Los subtipos de sprite **"heredarán"** las características de esta clase. Por ejemplo, como un Sprite tiene una coordenada X y una Y, también lo tendrá el OVNI, que es una subclase de Sprite. De igual modo, la nave se podrá "Dibujar" en pantalla, porque también es una subclase de Sprite.

- El propio juego también tendrá métodos adicionales, relacionados con la lógica del juego, como "ComprobarTeclas" (para ver qué teclas ha pulsado el usuario), "MoverElementos" (para actualizar el movimiento de los elementos que deban moverse por ellos mismos, como los enemigos o el OVNI), "ComprobarColisiones" (para ver si dos elementos chocan, como un disparo y un enemigo, y actualizar el estado del juego según corresponda), o "DibujarElementos" (para mostrar en pantalla todos los elementos actualizados).

Faltan detalles, pero no es un mal punto de partida. A partir de este diagrama podríamos crear un "esqueleto" de programa que compilase correctamente pero aún "no hiciese nada", y entonces comenzaríamos a repartir trabajo: una persona se podría encargar de crear la pantalla de bienvenida, otra de la lógica del juego, otra del movimiento del bloque de enemigos, otra de las peculiaridades de cada tipo de enemigo, otra del OVNI...

Nosotros no vamos a hacer proyectos tan grandes (al menos, no todavía), pero sí empezaremos a crear proyectos sencillos en los que colaboren varias clases, que permitan sentar las bases para proyectos más complejos, y también entender algunas peculiaridades de los temas que veremos a continuación, como el manejo de ficheros en C#.

Como curiosidad, cabe mencionar que en los proyectos grandes es habitual usar **herramientas gráficas** que nos ayuden a visualizar las clases y las relaciones que existen entre ellas, como hemos hecho para el Space Invaders. También se puede dibujar directamente en papel para aclararnos las ideas, pero el empleo de herramientas informáticas tiene varias ventajas adicionales:

- Podemos "pinchar y arrastrar" para recolocar las clases, y así conseguir más legibilidad o dejar hueco para nuevas clases que hayamos descubierto que también deberían ser parte del proyecto.
- Algunas de estas herramientas gráficas permiten generar automáticamente un esqueleto del programa, que nosotros rellenaremos después con los detalles de la lógica 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 (que es lo que a nosotros nos interesa en este momento), etc. Disponemos de herramientas gratuitas como **ArgoUML**, multiplataforma, que permite crear distintos tipos de diagramas UML y que permite generar el código correspondiente al esqueleto de nuestras clases, o **Dia**, también multiplataforma, pero de uso más genérico (para crear diagramas de cualquier tipo) y que no permite generar código por ella misma pero sí con la ayuda de Dia2code.

En los próximos ejemplos partiremos de una única clase en C#, para ampliar posteriormente esa estructura e ir creando proyectos más complejos.

Ejercicio propuesto:

(6.1.1) Piensa en un juego que conozcas, que no sea demasiado complejo (tampoco demasiado simple) y trata de hacer una descripción como la anterior y una descomposición en clases.

6.2. Objetos y clases en C#

Las **clases en C#** se definen de forma parecida a los registros (struct), sólo que ahora, además de variables (que representan sus detalles internos, y que llamaremos sus "atributos"), también incluirán funciones (las acciones que puede realizar ese objeto, que llamaremos sus "métodos"). Atributos y métodos formarán parte de "un todo", en vez de estar separados en distintas partes del programa. Esto es lo que se conoce como "**Encapsulación**".

Así, una clase "Puerta" se podría declarar así:

```
public class Puerta
{
    int ancho;        // Ancho en centímetros
    int alto;         // Alto en centímetros
    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"), muestre su estado, la abra y vuelva a mostrar su estado:

```

// Ejemplo_06_02a.cs
// Primer ejemplo de clases
// Introducción a C#, por Nacho Cabanes

using System;

public class Puerta
{
    int ancho;      // Ancho en centímetros
    int alto;       // Alto en centímetros
    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 Ejemplo_06_02a
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos 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 es el nuevo objetos con el que vamos a practicar.
- La clase "Ejemplo_06_02a", que representa a nuestra aplicación.

(Nota: al compilar, puede que obtengas algún "Aviso" -warning- que te dice que has declarado "alto", "ancho" y "color", pero no las estás utilizando; no es importante por ahora, puedes ignorar ese aviso).

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# (pero no en todos los 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, "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" desde el programa principal, 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, no lo será desde Ejemplo_06_02a).
- 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.

Por ejemplo, para conocer y modificar los valores del "ancho" de una puerta, podríamos crear un método LeerAncho, que nos devolviera su valor, y un método CambiarAncho, que lo reemplazase por otro valor. No hay un convenio claro sobre cómo llamar a a estos métodos en español, por lo que es frecuente usar las palabras inglesas "Get" y "Set" para leer y cambiar un valor, respectivamente. Así, 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;
}
```

Así, una nueva versión del programa, que incluya ejemplos de Get y Set, podría ser:

```
// Ejemplo_06_02b.cs
// Clases, get y set
// Introducción a C#, por Nacho Cabanes

using System;

public class Puerta
{
    int ancho;      // Ancho en centímetros
    int alto;       // Alto en centímetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public int GetAncho()
    {
        return ancho;
    }

    public void SetAncho(int nuevoValor)
    {
        ancho = nuevoValor;
    }

    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 Ejemplo_06_02b
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.SetAncho(80);
        p.MostrarEstado();
    }
}
```

```

    }
}

```

También puede desconcertar que en "Main" aparezca la palabra "**static**", mientras que no lo hace en los métodos de la clase "Puerta". Veremos el motivo un poco más adelante, pero de momento perderemos la costumbre de escribir "static" antes de cada función: a partir de ahora, **sólo Main será "static"**.

Ejercicios propuestos:

(6.2.1) Crea 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. Crea también una clase llamada PruebaPersona. Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre a cada uno y les pedirá que saluden.

(6.2.2) Tras leer la descripción de Space Invaders que vimos en el apartado anterior, crea una clase Juego, que sólo contenga un método Lanzar, void, sin parámetros, que escriba en pantalla "Bienvenido a Console Invaders. Pulse Intro para salir" y se parará hasta que el usuario pulse Intro. Prepara también un Main (en la misma clase), que cree un objeto de la clase Juego y lo lance.

(6.2.3) Para guardar información sobre libros, vamos a comenzar por crear una clase "Libro", que contendrá atributos "autor", "titulo", "ubicacion" (todos ellos strings) y métodos Get y Set adecuados para leer su valor y cambiarlo. Prepara también un Main (en la misma clase), que cree un objeto de la clase Libro, dé valores a sus tres atributos y luego los muestre.

(6.2.4) Crea una clase "Coche", con atributos "marca" (texto), "modelo" (texto), "cilindrada" (número entero), potencia (número real). No hace falta que crees un Main de prueba.

6.3. Proyectos a partir de varios fuentes

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).

Es recomendable (aunque 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".

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 tendría 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
```

En general, esto no lo podremos hacer de forma sencilla desde editores como Notepad++ y Geany: **Notepad++** tiene un menú "Ejecutar", que nos permite teclear una orden como la anterior, pero no supone ninguna gran ganancia; **Geany** permite crear "proyectos" formados por varios fuentes, pero no resulta una tarea especialmente cómoda. Por eso, veremos un primer ejemplo de cómo compilar desde "línea de comandos" y luego lo haremos con un entorno mucho más avanzado y que usaremos con frecuencia a partir de ahora: Visual Studio.

Vamos a **dividir en dos fuentes** el último ejemplo y a ver cómo se compilaría. La primera clase podría ser ésta:

```
// Puerta.cs
// Clases, get y set
// Introducción a C#, por Nacho Cabanes

using System;

public class Puerta
{
    int ancho;      // Ancho en centímetros
    int alto;       // Alto en centímetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public int GetAncho()
    {
        return ancho;
    }
}
```



```

    }

    public void SetAncho(int nuevoValor)
    {
        ancho = nuevoValor;
    }

    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_06_03a.cs
// Usa la clase Puerta
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_06_03a
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.SetAncho(80);
        p.MostrarEstado();
    }
}

```

Y lo compilaríamos con:

```
mcs ejemplo06_03a.cs puerta.cs -out:ejemploPuerta.exe
```

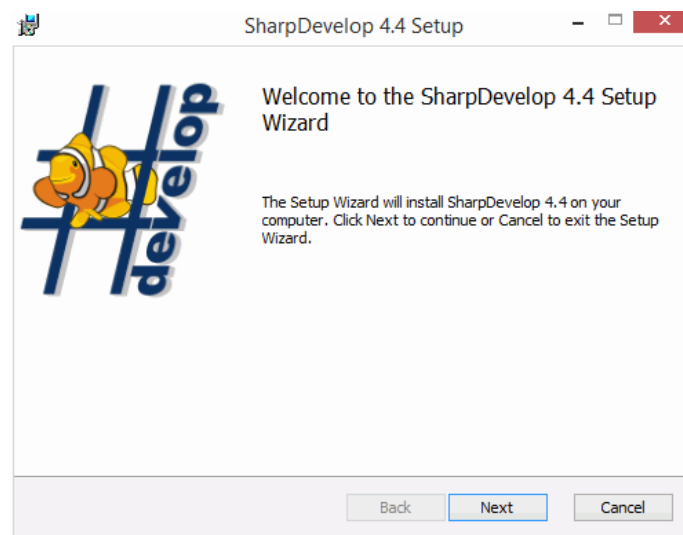
Aun así, para estos proyectos formados por varias clases, lo ideal es usar algún **entorno más avanzado**, como SharpDevelop (que además es gratuito) o VisualStudio (del que existe una versión gratuita, conocida como Express), que nos

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...

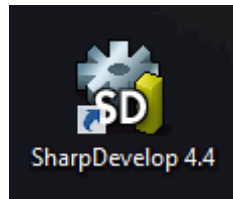
El entorno de desarrollo más conocido para crear programas de una cierta complejidad es Visual Studio, de Microsoft, que en su versión Express incluye todo lo que un programador novel como nosotros puede necesitar. Aun así, como puede requerir un equipo relativamente potente, vamos a comenzar por ver una alternativa muy similar, pero algo más sencilla, llamada **SharpDevelop**, disponible para Windows.

Comenzamos por descargar el fichero de instalación del entorno, desde su página oficial (<http://www.icsharpcode.net/OpenSource/SD/Download/>). La versión 4.4, para las versiones 2.0 a 4.5.1 de la plataforma .Net, ocupa unos 15 Mb. En el momento de escribir este texto también existe una versión 5.0, pero todavía está en "fase beta" (pruebas previas a la versión definitiva, así que puede contener algún error).

La instalación comenzará al hacer doble clic en el fichero descargado. Deberíamos ver una ventana parecida a ésta:

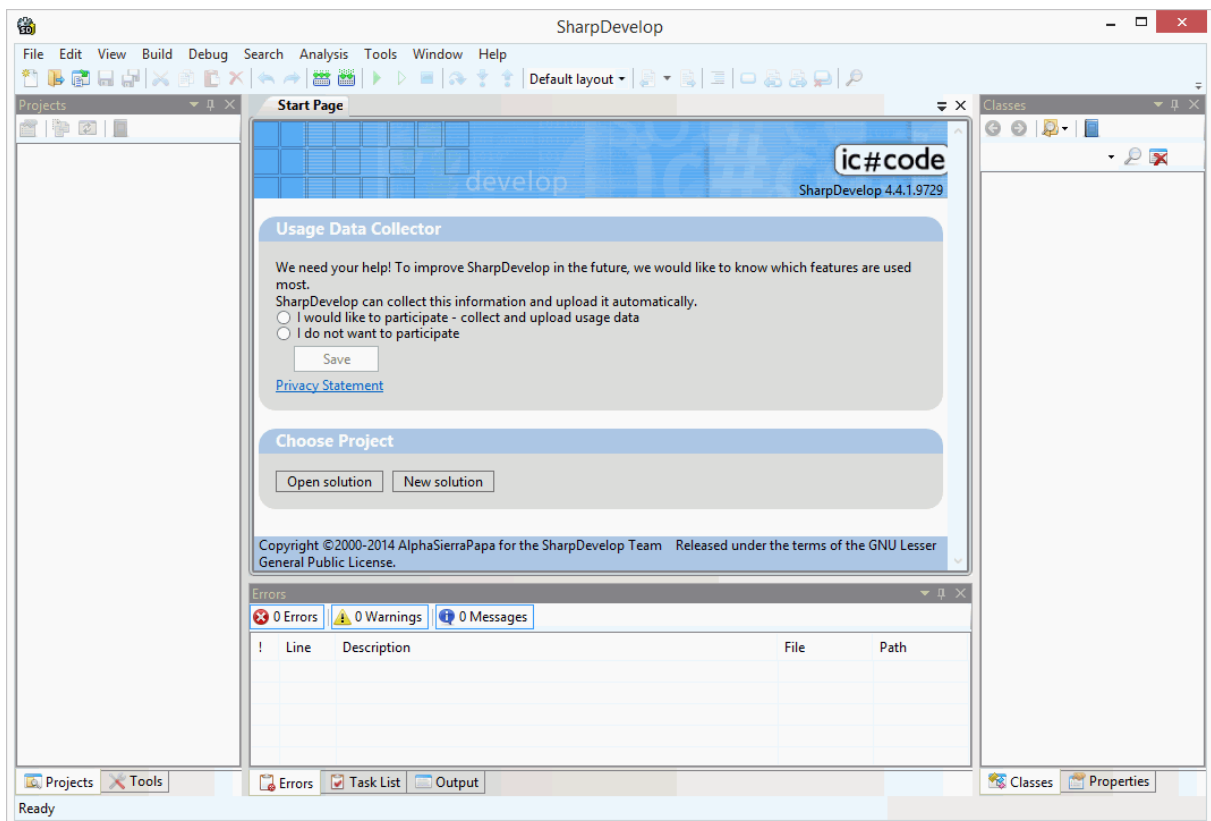


Como es habitual, el siguiente paso será aceptar el contrato de licencia, después deberemos decir en qué carpeta queremos instalarlo, comenzará a copiar archivos y al cabo de un instante, tendremos un nuevo icono en nuestro escritorio:

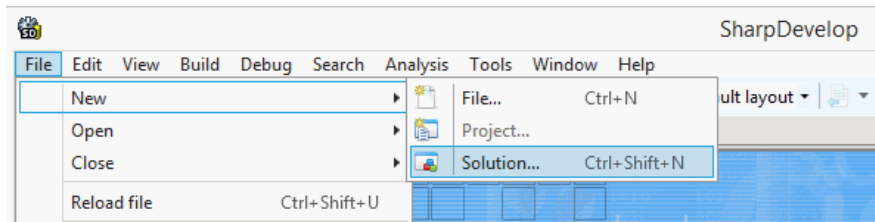


La instalación debería ser muy sencilla en Windows Vista y superiores, pero en Windows XP quizá necesite que instalemos una versión reciente de la plataforma .Net (se puede descargar gratuitamente desde la web de Microsoft).

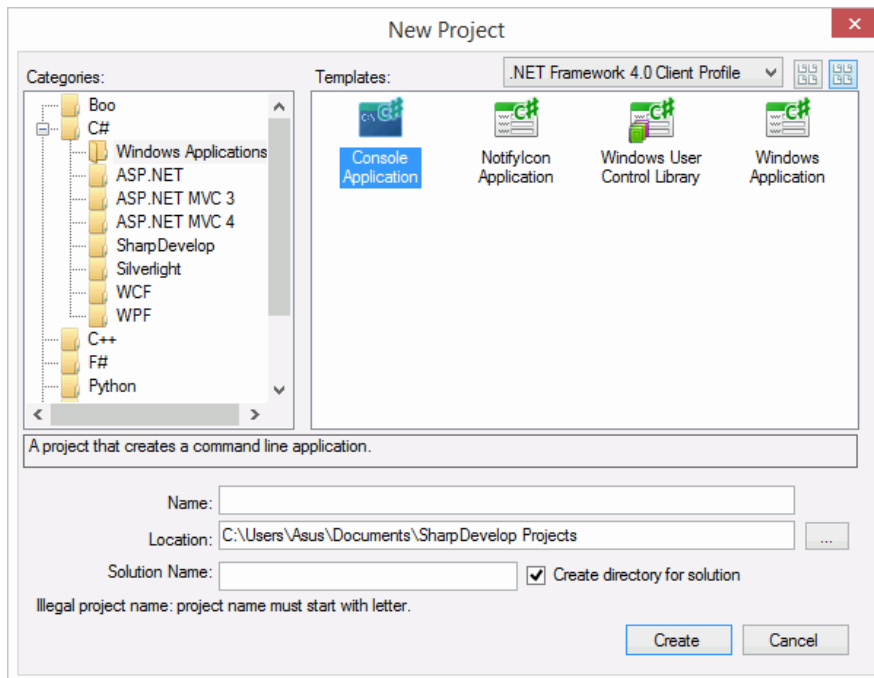
Cuando hagamos doble clic en nuestro nuevo icono, veremos la pantalla principal de SharpDevelop, que nos muestra la lista de los últimos proyectos ("soluciones") que hemos realizado, y nos permite crear uno nuevo:



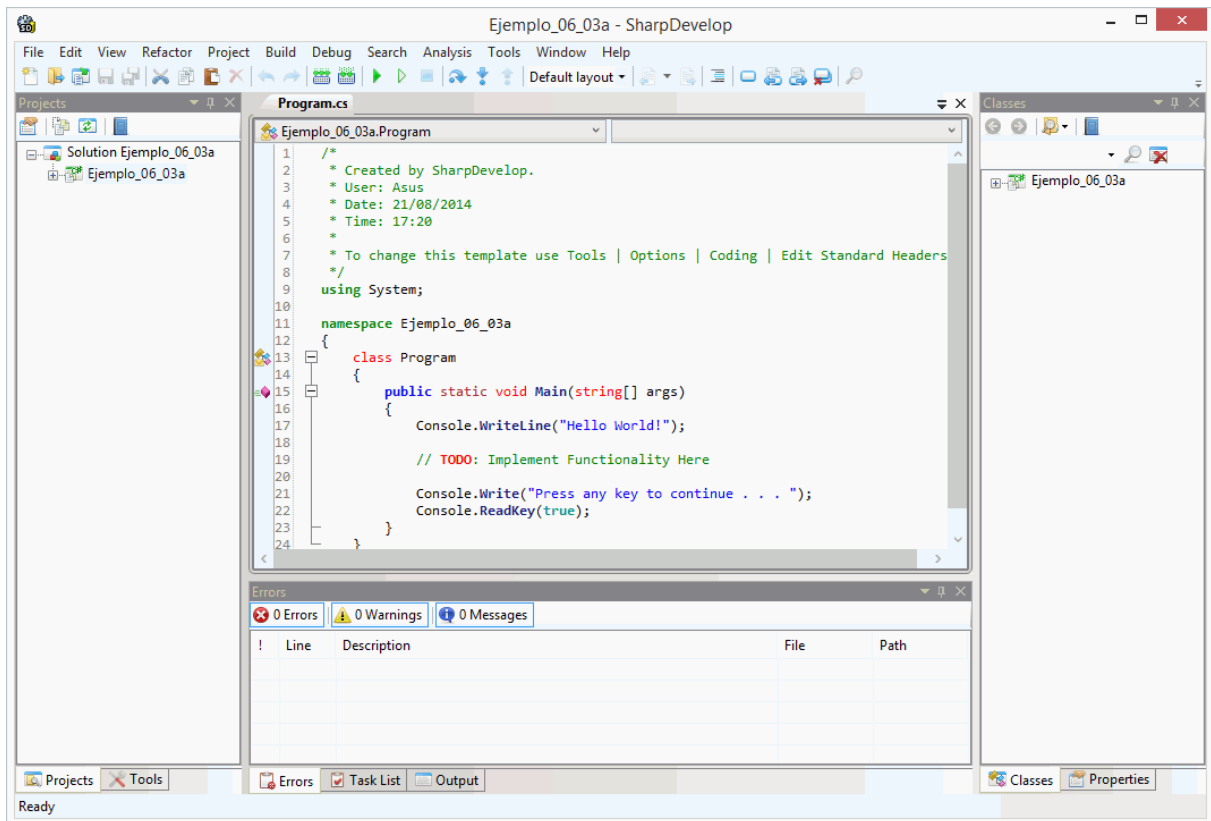
En nuestro caso, comenzaremos por crear una "Nueva solución", desde el menú "File", en la opción "New Solution":



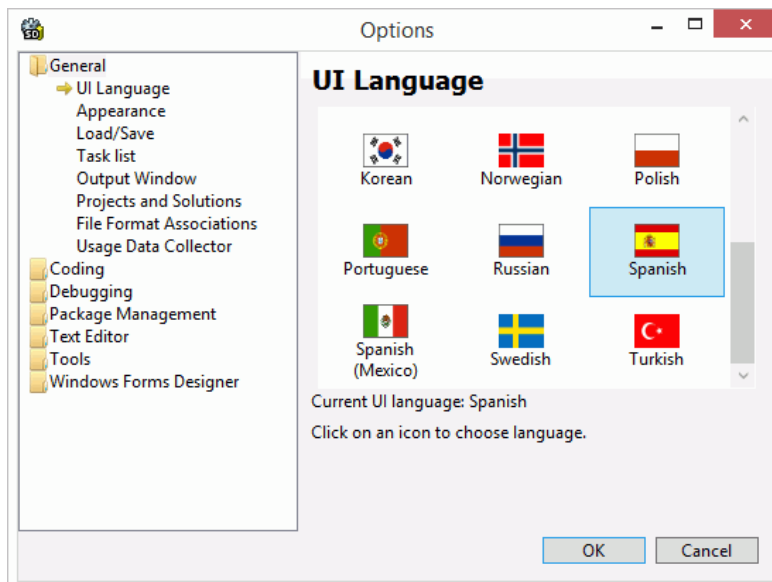
Se nos mostrará los tipos de proyectos para los que se nos podría crear un esqueleto vacío que después iríamos rellenando:



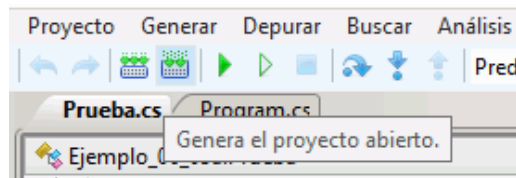
De estos tipos, el único que conocemos es una "Aplicación de Consola" (Console Application) en C#. Debemos escribir también el nombre, y aparecerá un esqueleto de aplicación que nosotros sólo tendríamos que completar:



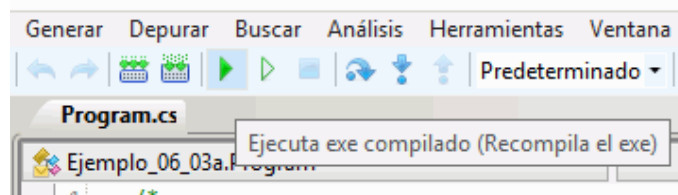
Podemos cambiar el idioma del interfaz, para que todos los menús y opciones aparezcan **en español**. Lo haremos desde "Options" (opciones), al final del menú "Tools" (herramientas):



Cuando hayamos terminado de realizar nuestros cambios, podemos compilar el programa con el botón "Generar":

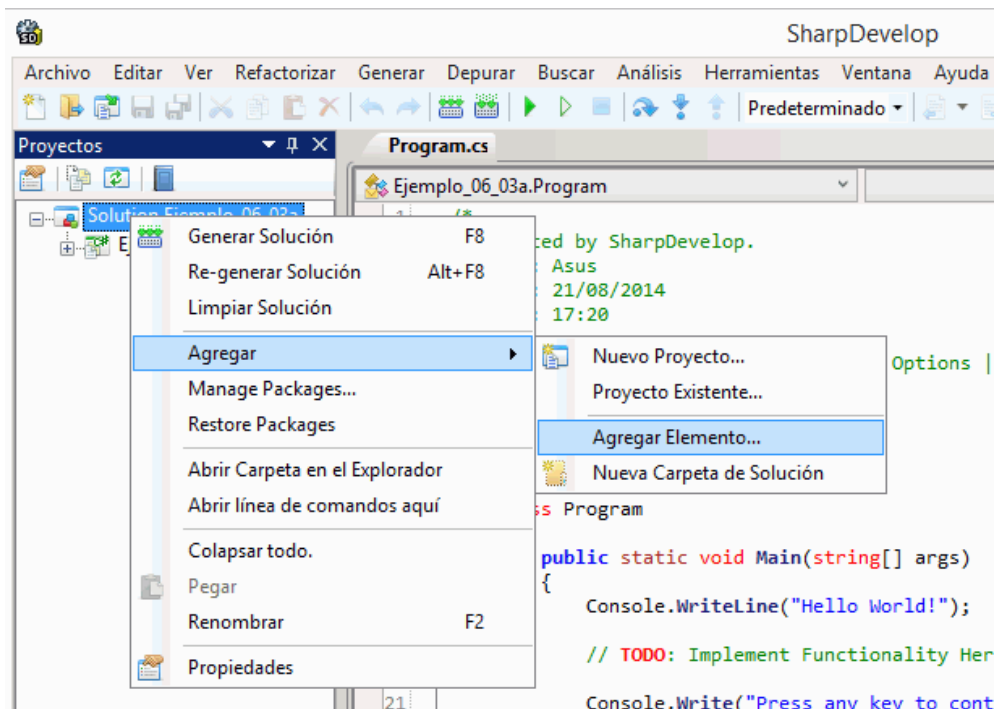


Si hubiera algún error, se nos avisaría en la parte inferior de la pantalla, y se subrayarían en rojo las líneas correspondientes de nuestro programa; si todo ha ido bien, podremos ejecutar nuestro programa para verlo funcionando:

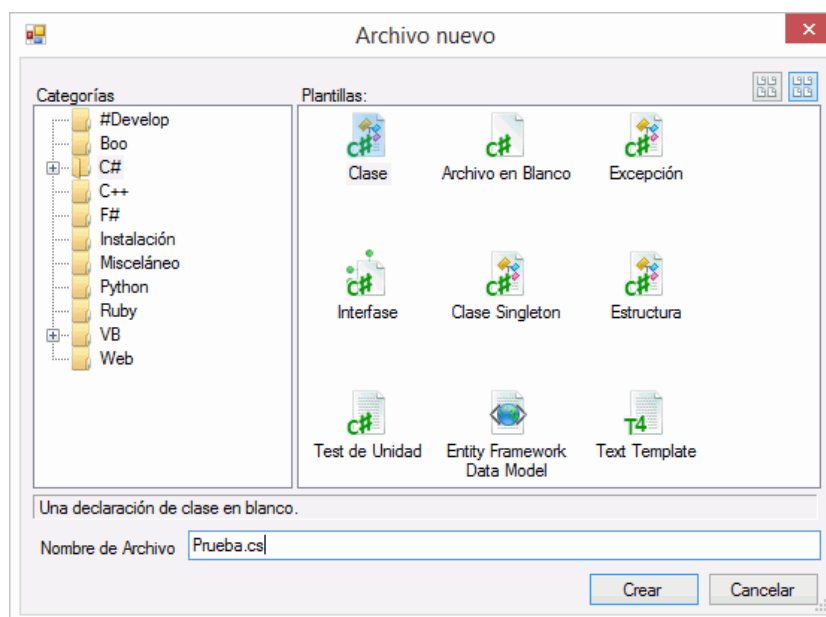


(Si la ventana de nuestro programa se cierra tan rápido que no tenemos tiempo de leerla, nos puede interesar añadir provisionalmente una línea `ReadLine()` al final del fuente, para que éste se detenga hasta que pulsemos la tecla Intro).

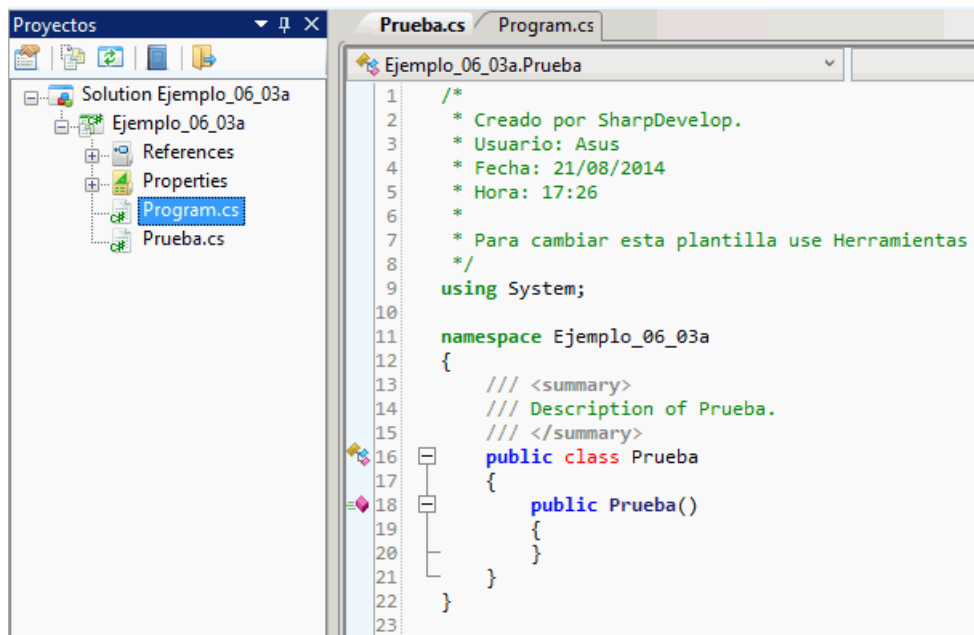
Así prepararíamos y lanzaríamos un programa formado por un solo fuente. Si se trata de varios fuentes, basta con ir añadiendo nuevas clases al proyecto. Lo conseguimos pulsando el botón derecho sobre el nombre del proyecto (en la ventana izquierda, "Proyectos") y escogiendo las opciones Agregar / Agregar 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.

Si nuestro equipo es moderno, posiblemente nos permitirá usar **Visual Studio** con una cierta fluidez. Instalar y emplear este entorno es muy similar a lo que hemos visto. La versión Express, que es gratuita para uso personal, se puede descargar de:

<http://www.visualstudio.com/es-es/products/visual-studio-express-vs>

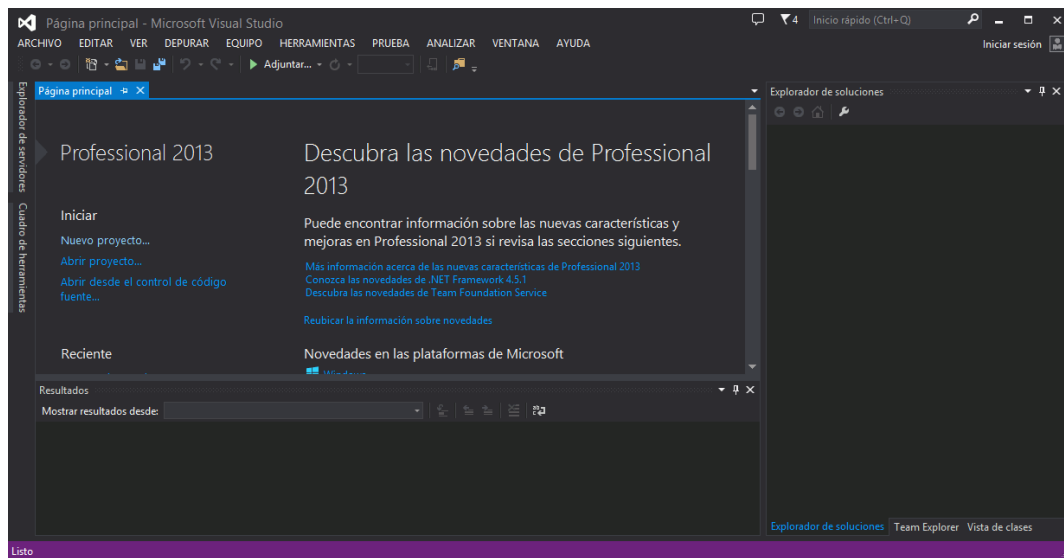
(En nuestro caso, se trataría de la versión Express para escritorio de Windows - Windows Desktop-)

La instalación es muy similar a la de SharpDevelop: poco más que descargar el fichero, hacer doble clic para comenzar (si descargamos y esperar un momento a que se complete el proceso (puede ser necesario tener que reiniciar el ordenador en algún punto intermedio, si nuestra versión de la plataforma "punto net" no es lo suficientemente moderna).

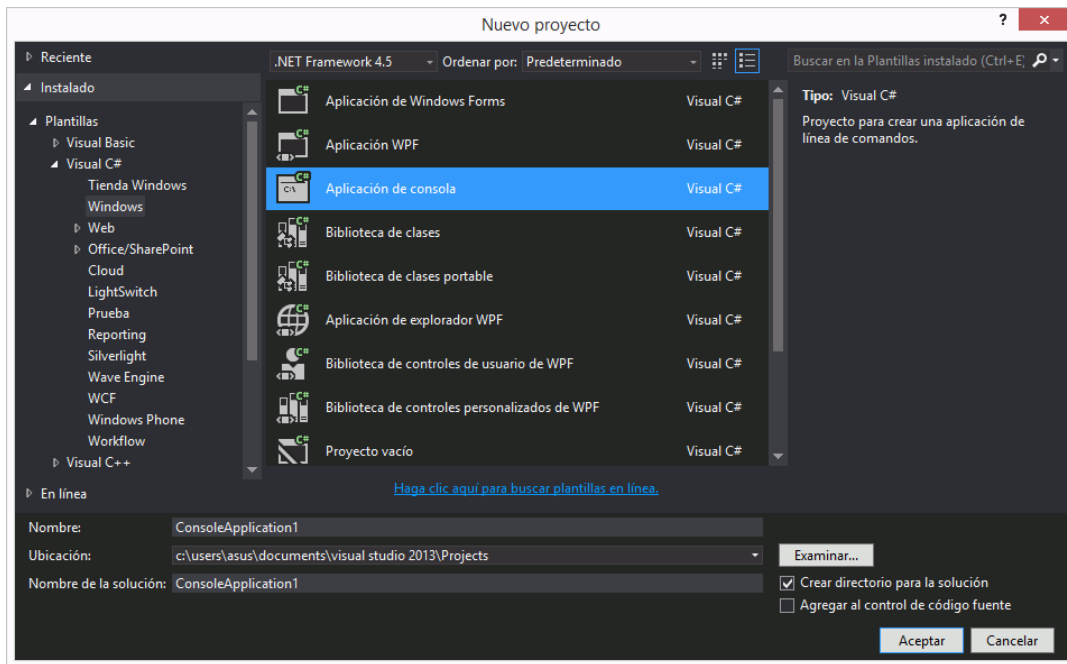
Si escogemos el instalador basado en Web, descargaremos un pequeño fichero al principio, y necesitaremos estar conectados a Internet durante la instalación para que se reciban los ficheros necesarios. Si, por el contrario, descargamos la "imagen ISO de DVD", deberemos grabarla en un DVD o extraerla con 7-zip o WinRAR para acceder a los ficheros de instalación.

Si eres estudiante y tu centro de estudios está adscrito a programas como la "Academic Alliance" de Microsoft (MSDNAA), posiblemente tendrás acceso gratuito también a otras versiones más potentes de Visual Studio, como Professional y Ultimate, con más características adicionales (que no necesitarás siendo un principiante, como las pruebas de rendimiento o automatización de tests para interfaces de usuario).

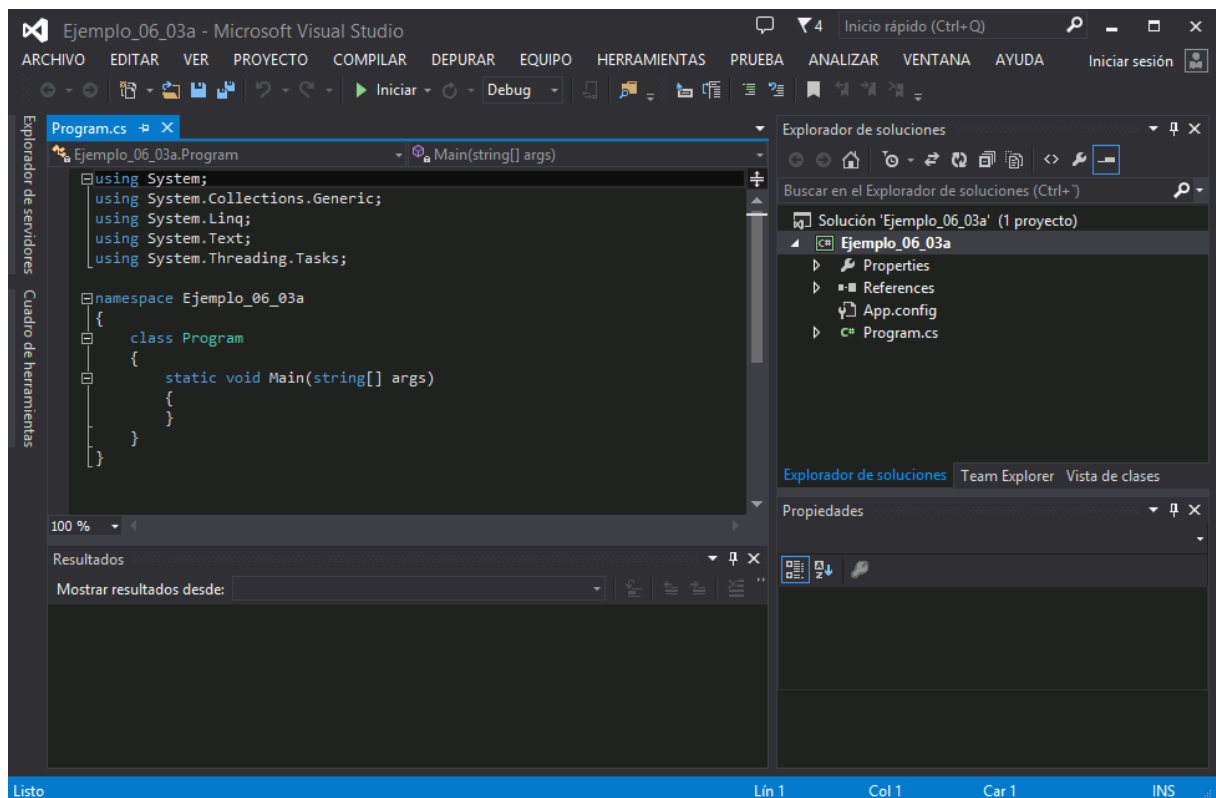
La pantalla principal, una vez completada la instalación, debería ser similar a ésta (para Visual Studio 2013; puede ser ligeramente distinta en otras versiones o si escogemos otras paletas de colores):



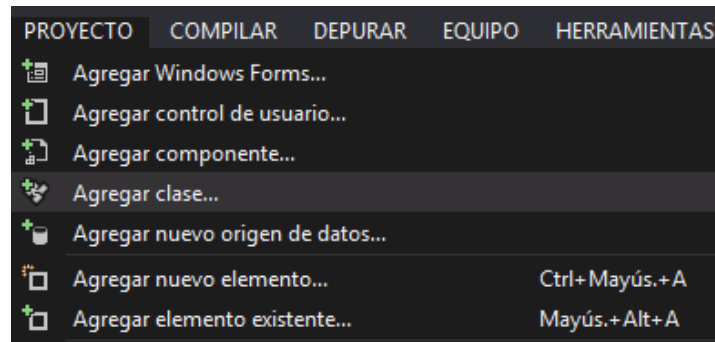
En ella tenemos la opción de crear un "Nuevo proyecto". De entre los tipos de proyectos existentes, escogeríamos crear una "Aplicación de consola":



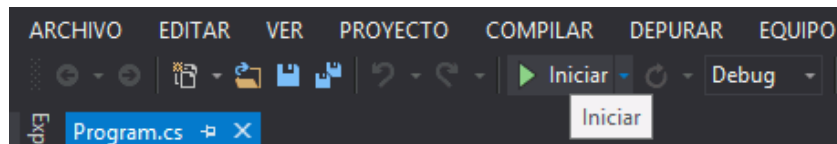
En la parte inferior escribiremos el nombre de nuestro proyecto, y pulsaremos el botón "Aceptar", con lo que debería aparecer un esqueleto de programa vacío:



Si nuestro programa va a estar formado por varios fuentes, podríamos añadir más clases usando la ventana de la derecha ("Explorador de soluciones") o la opción "Agregar clase..." del menú "Proyecto":



Que, al igual que en el caso de SharpDevelop, nos mostraría el esqueleto de un fuente, que en este caso no contiene "Main". Cuando el programa esté listo, podemos lanzarlo haciendo clic en el botón "Iniciar":



Hay alguna **diferencia** más en el manejo normal de Visual Studio Express, comparado con el de SharpDevelop, que conviene tener en cuenta:

- No es necesario compilar para saber los errores: se irán marcando en rojo las líneas incorrectas **a medida que las tecleemos**.
- Según la versión de Visual Studio, puede ser que **"compile desde memoria"**: si pulsamos el botón de Iniciar, nuestro proyecto **no se guarda automáticamente**, sino que deberemos usar explícitamente la opción de Guardar cuando nosotros lo deseemos. Si no lo hacemos, corremos el riesgo de que no se haya llegado a guardar nuestro proyecto, y perdamos el trabajo en caso de fallo del sistema operativo o del suministro de corriente eléctrica.
- La mayoría de versiones de Visual Studio permiten lanzar nuestro programa de modo que la ejecución se pause al terminar, igual que ocurre en Geany. Para conseguirlo, en vez de pulsar el botón "Iniciar" de la barra de herramientas, utilizaremos la combinación de teclas **Ctrl+F5**.

```

C:\WINDOWS\system32\cmd.exe
Valores iniciales...
Ancho: 0
Alto: 0
Color: 0
Abierta: False

Vamos a abrir...
Ancho: 80
Alto: 0
Color: 0
Abierta: True
Presione una tecla para continuar . . .

```

Quien trabaja con Linux o con Mac, no podrá emplear SharpDevelop ni VisualStudio, pero tiene una alternativa muy similar, desarrollada por el mismo equipo que ha creado Mono. Se trata de **MonoDevelop**, que se debería poder instalar apenas en un par de clics con el gestor de paquetes del sistema (por ejemplo, Synaptic en el caso de Ubuntu y de Linux Mint).

Ejercicio propuesto:

(6.3.1) Crea un proyecto con las clases Puerta y Ejemplo_06_03a. Comprueba que todo funciona correctamente.

(6.3.2) Modifica el fuente del ejercicio 6.2.1 (clase Persona), para dividirlo en dos ficheros: Crea 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. Crea 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.

(6.3.3) Crea un proyecto a partir de la clase Libro (ejercicio 6.2.3). El "Main" pasará a una segunda clase llamada "PruebaDeLibro" y desaparecerá de la clase Libro.

(6.3.4) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.2.2): crea un proyecto para Visual Studio o SharpDevelop. Además de la clase "Juego", crea una clase "Bienvenida" y una clase "Partida". El método "Lanzar" de la clase Juego, ya no escribirá nada en pantalla, sino que creará un objeto de la clase "Bienvenida" y lo lanzará y luego un objeto de la clase "Partida" y lo lanzará. El método Lanzar de la clase Bienvenida escribirá en pantalla "Bienvenido a Console Invaders. Pulse Intro para jugar". El método Lanzar de la clase Partida escribirá en pantalla "Ésta sería la

pantalla de juego. Pulse Intro para salir" y se parará hasta que el usuario pulse Intro.

(6.3.5) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.3.4): El método Lanzar de la clase Bienvenida escribirá en pantalla "Bienvenido a Console Invaders. Pulse Intro para jugar o ESC para salir". Puedes comprobar si se pulsa ESC con "ConsoleKeyInfo tecla = Console.ReadKey(); if (tecla.Key == ConsoleKey.Escape) salir = true;". El código de la tecla Intro es " ConsoleKey.Enter". También puedes usar "Console.Clear();" si quieres borrar la pantalla. Añade un método "GetSalir" a la clase Bienvenida, que devuelva "true" si el usuario ha escogido Salir o "false" si ha elegido Jugar. El método Lanzar de la clase Juego repetirá la secuencia Bienvenida-Partida hasta que el usuario escoja Salir.

(6.3.6) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.3.5): Crea una clase Nave, con atributos "x" e "y" (números enteros, "x" de 0 a 1023 e "y" entre 0 y 767, pensando en una pantalla de 1024x768), e imagen (un string formado por dos caracteres, como "\/"). También tendrá un método MoverA(nuevaX, nuevaY) que lo mueva a una nueva posición, y un método Dibujar, que muestre esa imagen en pantalla (como esta versión es para consola, la X tendrá que rebajarse para que tenga un valor entre 0 y 79, y la Y entre 0 y 24). Puedes usar Console.SetCursorPosition(x,y) para situarte en unas coordenadas de pantalla. Crea también clase Enemigo, con los mismos atributos. Su imagen podría ser "[]". El método Lanzar de la clase Partida creará una nave en las coordenadas (500, 600) y la dibujará, creará un enemigo en las coordenadas (100, 80) y lo dibujará, y finalmente esperará a que el usuario pulse Intro para terminar la falsa sesión de juego.

(6.3.7) Crea un proyecto a partir de la clase Coche (ejercicio 6.2.4): además de la clase Coche, existirá una clase PruebaDeCoche, que contendrá la función "Main", que creará un objeto de tipo coche, pedirá al usuario su marca, modelo, cilindrada y potencia, y luego mostrará en pantalla el valor de esos datos.

6.4. La herencia

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:

```
// Porton.cs
// Clase que hereda de Puerta
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```

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.

En este ejemplo, la clase Puerta no cambiaría nada.

No tenemos por qué heredar todo tal y como era; 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 (si no incluimos la palabra "new", el compilador mostrará un "warning", pero dará el programa como válido; más adelante veremos que existe una alternativa a "new" y en qué momento será adecuado usar cada una de ellas.

```

public new void MostrarEstado()
{
    Console.WriteLine("Portón.");
    Console.WriteLine("Bloqueada: {0}", bloqueada);
}

```

Puedes observar que ese "MostrarEstado" no dice nada sobre el ancho ni el alto del portón. En el próximo apartado veremos cómo acceder a esos datos.

Un programa de prueba, que ampliase el anterior para incluir un "portón", podría ser:

```
// Ejemplo_06_04a.cs
```

```
// Portón, que hereda de Puerta
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_06_04a
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.SetAncho(80);
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Ahora el portón...");
        Porton p2 = new Porton();
        p2.SetAncho(300);
        p2.Bloquear();
        p2.MostrarEstado();
    }
}
```

Y su resultado sería:

```
Valores iniciales...
Ancho: 0
Alto: 0
Color: 0
Abierta: False
```

```
Vamos a abrir...
Ancho: 80
Alto: 0
Color: 0
Abierta: True
```

```
Ahora el portón...
Portón.
Bloqueada: True
```

Ejercicios propuestos:

(6.4.1) Crea un proyecto con las clases Puerta, Portón y Ejemplo_06_04a. Prueba que todo funciona correctamente.

(6.4.2) Crea una variante ampliada del ejercicio 6.3.2. En ella, la clase `Persona` no cambia. Se creará una nueva clase `PersonalInglesa`, en el fichero `"personalInglesa.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 `PersonalInglesa`, les asignará un nombre, les pedirá que saluden y pedirá a la persona inglesa que tome té.

(6.4.3) Amplía el proyecto del ejercicio 6.3.3 (Libro): crea una clase `"Documento"`, de la que `Libro` heredarán todos sus atributos y métodos. Ahora la clase `Libro` contendrá sólo un atributo `"paginas"`, número entero, con sus correspondientes `Get` y `Set`.

(6.4.4) Amplía el esqueleto del `ConsoleInvaders` (ejercicio 6.3.6): Crea una clase `"Sprite"`, de la que heredarán `"Nave"` y `"Enemigo"`. La nueva clase contendrá todos los atributos y métodos que son comunes a las antiguas (todos los existentes, por ahora).

(6.4.5) Amplía el proyecto de la clase `Coche` (ejercicio 6.3.7): Crea una clase `"Vehiculo"`, de la que heredarán `"Coche"` y una nueva clase `"Moto"`. La clase `Vehiculo` contendrá todos los atributos y métodos que antes estaban en `Coche`, y tanto `Coche` como `Moto` heredarán de ella.

6.5. Visibilidad

Nuestro ejemplo todavía no funciona correctamente: los atributos de una `Puerta`, como el `"ancho"` y el `"alto"` estaban declarados como `"privados"` (es lo que se considera en C# si no decimos lo contrario), por lo que no son accesibles desde ninguna otra clase, ni siquiera desde `Porton`.

Hemos manejado con frecuencia la palabra `"public"`, para indicar que algo debe ser público, porque nuestras clases y su `"Main"` lo han sido siempre hasta ahora. Nos podríamos sentir tentados de declarar como `"public"` los atributos como el `"ancho"` y el `"alto"`, pero esa no es la solución más razonable, porque no queremos que sean accesibles desde cualquier sitio, debemos recordar la máxima de `"ocultación de detalles"`, que hace nuestros programas sean más fáciles de mantener.

Sólo querríamos 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". En general, será preferible usar "private" a no escribir nada, por legibilidad, para ayudar a detectar errores con mayor facilidad y como costumbre por si más adelante programamos en otros lenguajes, porque puede ocurrir que en otros lenguajes se considere público (en vez de privado) un atributo cuya visibilidad no hayamos indicado.

Así, un único 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_06_05a.cs
// Portón, que hereda de Puerta, con "protected"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_06_05a
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.SetAncho(80);
        p.MostrarEstado();

        Console.WriteLine();

        Console.WriteLine("Ahora el portón...");
    }
}
```

```

        Porton p2 = new Porton();
        p2.SetAncho(300);
        p2.Bloquear();
        p2.MostrarEstado();
    }

}

// -----

// Puerta.cs
// Clases, get y set
// Introducción a C#, por Nacho Cabanes

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 int GetAncho()
    {
        return ancho;
    }

    public void SetAncho(int nuevoValor)
    {
        ancho = nuevoValor;
    }

    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

// -----

// Porton.cs
// Clase que hereda de Puerta
// Introducción a C#, por Nacho Cabanes

public class Porton : Puerta

```

```

{
    bool bloqueada;

    public void Bloquear()
    {
        bloqueada = true;
    }

    public void Desbloquear()
    {
        bloqueada = false;
    }

    public new void MostrarEstado()
    {
        Console.WriteLine("Portón.");
        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);
    }
}

```

Ejercicios propuestos:

(6.5.1) Crea un proyecto a partir del ejemplo 06.05a, en el que cada clase esté en un fichero separado. Como podrás comprobar, ahora necesitarás un "using System" en cada fuente.

(6.5.2) Amplía las clases del ejercicio 6.4.2, creando un nuevo proyecto con las siguientes características: La clase Persona no cambia; la clase PersonalInglesa 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 PersonalItaliana, en el fichero "personalitaliana.cs", que deberá heredar las características de la clase "Persona", pero redefinir el método "Saludar", para que escriba en pantalla "Ciao"; crea también una clase llamada PruebaPersona3, en el fichero "pruebaPersona3.cs", que deberá contener sólo la función Main y creará un objeto de tipo Persona, dos de tipo PersonalInglesa, uno de tipo PersonalItaliana, les asignará un nombre, les pedirá que saluden y pedirá a la persona inglesa que tome té.

(6.5.3) Retoca el proyecto del ejercicio 6.4.3 (Libro): los atributos de la clase Documento y de la clase Libro serán "protegidos".

(6.5.4) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.4.4): Amplía la clase Nave con un método "MoverDerecha", que aumente su X en 10 unidades, y un "MoverIzquierda", que disminuya su X en 10 unidades. Necesitarás hacer que esos atributos sean "protected". El método Lanzar de la clase Partida no esperará hasta el usuario pulse Intro sin hacer nada, sino que ahora usará un do-while que

compruebe si pulsa ESC (para salir) o flecha izquierda o flecha derecha (para mover la nave: sus códigos son `ConsoleKey.LeftArrow` y `ConsoleKey.RightArrow`). Si se pulsan las flechas, la nave se moverá a un lado o a otro (con los métodos que acabas de crear). Al principio de cada pasada del `do-while` se borrará la pantalla (`"Console.Clear();"`).

(6.5.5) Mejora el proyecto de la clase `Coche` (ejercicio 6.4.5): todos los atributos serán "protegidos" y los métodos serán "públicos".

6.6. Constructores y destructores

Hemos visto que al declarar una clase, automáticamente 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, para leer información desde fichero, etc.

Un constructor 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 = 0xFFFFFF;
    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 = 0xFFFFFF;
    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_06_06a.cs
// Constructores
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_06_06a
{
    public static void Main()
    {
        Puerta p = new Puerta();
        Puerta p2 = new Puerta(90,220);

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine();
        Console.WriteLine("Vamos a abrir...");
        p.Abrir();
        p.MostrarEstado();

        Console.WriteLine();
        Console.WriteLine("Para la segunda puerta...");
        p2.MostrarEstado();
    }
}

// -----

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 Puerta()
    {
        ancho = 100;
        alto = 200;
        color = 0xFFFFFF;
        abierta = false;
    }

    public Puerta(int an, int al)
    {

```

```

        ancho = an;
        alto = al;
        color = 0xFFFFF;
        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

```

Nota: al igual que existen los "constructores", también podemos crear 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). Nuestros programas son tan sencillos que todavía no los necesitarán.

Un destructor se creará con el mismo **nombre** que la clase y que el constructor, pero precedido por el símbolo "~", y no tiene tipo de retorno, ni parámetros, ni especificador de acceso ("public" ni ningún otro), como ocurre en este ejemplo:

```

~Puerta()
{
    // Liberar memoria
    // Cerrar ficheros
}

```

Ejercicios propuestos:

(6.6.1) Ampliar las clases del ejercicio 6.5.2, para que todas ellas contengan constructores. Los constructores de casi todas las clases estarán vacíos, excepto del de PersonalInglesa, que prefijará su nombre a "John". Crea también un constructor alternativo para esta clase que permita escoger cualquier otro nombre.

(6.6.2) Amplía el proyecto del ejercicio 6.5.3 (Libro): la clase Libro tendrá un constructor que permita dar valores al autor, el título y la ubicación.

(6.6.3) Amplía el esqueleto del ConsoleInvaders (ejercicio 6.5.4): La clase Enemigo tendrá un constructor, sin parámetros, que prefijará su posición inicial. El constructor de la clase Nave recibirá como parámetros las coordenadas X e Y iniciales, para que se puedan cambiar desde el cuerpo del programa. Elimina las variables xNave e yNave de la clase Partida, que ya no serán necesarias.

(6.6.4) Mejora el proyecto de la clase Coche (ejercicio 6.5.5): añade un atributo "cantidadDeRuedas" a la clase Vehiculo, junto con sus Get y Set. El constructor de la clase Coche le dará el valor 4 y el constructor de la clase Moto le dará el valor 2.

6.7. Polimorfismo y sobrecarga

El concepto de "**polimorfismo**" se refiere a que una misma función (un método) puede tener varias formas, ya sea porque reciba distintos tipos de parámetros y/o en distinta cantidad, o porque incluso se aplique a distintos objetos.

Existen dos tipos especialmente importantes de polimorfismo:

- En nuestro último ejemplo (06_06a), los 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 "**sobrecarga**" (también conocida como "polimorfismo ad-hoc"). Es un tipo de polimorfismo en el que el compilador sabe en tiempo de compilación a qué método se debe llamar.
- El caso opuesto es el "**polimorfismo puro**", en el que un mismo método se aplica a distintos objetos de una misma jerarquía (como el "MostrarEstado" del ejemplo 06_05a, que se puede aplicar a una puerta o a un portón), y en ese caso el compilador puede llegar a no ser capaz de saber en tiempo de compilación a qué método se debe llamar, y lo tiene que descubrir en tiempo de ejecución. Es algo que nos encontraremos un poco más adelante, cuando hablemos de "funciones virtuales".

Ejercicios propuestos:

(6.7.1) A partir de las clases del ejercicio 6.6.1, añade a la clase "Persona" un nuevo método Saludar, que reciba un parámetro, que será el texto que debe decir esa persona cuando salude.

(6.7.2) Amplía el proyecto del ejercicio 6.6.2 (Libro): la clase Libro tendrá un segundo constructor que permita dar valores al autor y el título, pero no a la ubicación, que tomará el valor por defecto "No detallada".

(6.7.3) Amplía el esqueleto del ConsoleInvaders (6.6.3): La clase Nave tendrá un segundo constructor, sin parámetros, que prefijará su posición inicial a (500,600). La clase Enemigo tendrá un segundo constructor, con parámetros X e Y, para poder colocar un enemigo en cualquier punto desde Main.

(6.7.4) Crea dos nuevos métodos en la clase Vehiculo (ejercicio 6.6.4): uno llamado Circular, que fijará su "velocidad" (un nuevo atributo) a 50, y otro Circular(v), que fijará su velocidad al valor que se indique como parámetro.

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_06_08a.cs
// Orden de llamada a los constructores
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_06_08a
{
    public static void Main()
    {
        Animal a1      = new Animal();
        Console.WriteLine();

        GatoSiames a2 = new GatoSiames();
        Console.WriteLine();

        Perro a3      = new Perro();
        Console.WriteLine();

        Gato a4      = new Gato();
        Console.WriteLine();
    }
}

// -----

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");
    }
}

```

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

Ejercicios propuestos:

(6.8.1) Crea 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.

(6.8.2) Crea una variante del proyecto Libro (ejercicio 6.7.2) en la que el constructor de Documento escriba en pantalla "Creando documento" y el constructor de Libro escriba en pantalla "Creando libro". Comprueba su funcionamiento.

(6.8.3) Crea una versión alternativa del esqueleto del ConsoleInvaders (6.7.3) en la que el constructor de Sprite escriba en pantalla "Creando sprite" y los constructores de Nave escriba en pantalla "Creando nave en posición prefijada" o "Creando nave en posición indicada por el usuario", según el caso. Comprueba su funcionamiento.

(6.8.4) Crea una versión alternativa de las clases Vehiculo, Coche, Moto (6.7.4), que te avise del momento en que se entra a cada constructor. Crea un programa de prueba que defina un coche y una moto, y comprueba su funcionamiento.

7. Utilización avanzada de clases

7.1. La palabra "static"

Desde un principio, nos hemos encontrado con que "Main" siempre iba acompañado de la palabra "static". Lo mismo ocurría con las funciones que creamos en el tema 5. 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 tener el valor 4 para cualquier objeto que pertenezca a esa clase (cualquier coche). Por eso, se podría declarar como "static". En el mundo real, esto es muy poco habitual, y por eso casi nunca usaremos atributos "static" (por ejemplo, no todos los coches que podamos encontrar tendrán 4 ruedas, aunque esa sea la cantidad más frecuente).
- 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:

```
// Ejemplo_07_01a.cs
// Métodos "static"
// Introducción a C#, por Nacho Cabanes

using System;

public class Hardware
{
    public static void BorrarPantalla()
    {
        for (byte i = 0; i < 25; i++)
            Console.WriteLine();
    }
}

public class Ejemplo_07_01a
{
    public static void Main()
    {
        Console.WriteLine("Pulsa Intro para borrar");
        Console.ReadLine();
    }
}
```

```

        Hardware.BorrarPantalla();
        Console.WriteLine("Borrado!");
    }
}

```

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 directamente desde "Main" (como hicimos en el tema 5, cuando aún no conocíamos las "clases").
- Declarar un objeto de la clase correspondiente, y entonces sí podremos acceder a sus métodos desde "Main":

```

// Ejemplo_07_01b.cs
// Alternativa a 07_01a, sin métodos "static"
// Introducción a C#, por Nacho Cabanes

using System;

public class Hardware
{
    public void BorrarPantalla()
    {
        for (byte i = 0; i < 25; i++)
            Console.WriteLine();
    }
}

public class Ejemplo_07_01b
{
    public static void Main()
    {
        Console.WriteLine("Pulsa Intro para borrar");
        Console.ReadLine();
        Hardware miPantalla = new Hardware();
        miPantalla.BorrarPantalla();
        Console.WriteLine("Borrado!");
    }
}

```

Ejercicios propuestos:

(7.1.1) Amplía el ejemplo 07_01a con un función "static" llamada "EscribirCentrado", que escriba centrado horizontalmente el texto que se le indique como parámetro.

(7.1.2) Amplía el ejemplo 07_01b con un función llamada "EscribirCentrado", que escriba centrado horizontalmente el texto que se le indique como parámetro. Al contrario que en el ejercicio 7.1.1, esta versión no será "static".

(7.1.3) Crea una nueva versión del ejercicio 5.2.3 (base de datos de ficheros, descompuesta en funciones), en la que los métodos y variables no sean "static".

7.2. Arrays de objetos

Es muy frecuente que no nos baste con tener un único objeto de una clase, sino que necesitemos manipular varios objetos pertenecientes a la misma clase. En ese caso, podemos almacenar todos ellos en un "array".

Deberemos usar "new" dos veces: primero para reservar memoria para el array, y luego para cada uno de los elementos. Por ejemplo, podríamos partir del ejemplo del apartado 6.8 y crear un array de 5 perros así:

```
Perro[] misPerros = new Perro[5];
for (byte i = 0; i < 5; i++)
    misPerros[i] = new Perro();
```

Un fuente completo de ejemplo podría ser

```
// Ejemplo_07_02a.cs
// Array de objetos
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_02a
{
    public static void Main()
    {
        Perro[] misPerros = new Perro[5];
        for (byte i = 0; i < 5; i++)
            misPerros[i] = new Perro();
    }
}

// -----

public class Animal
{
    public Animal()
    {
        Console.WriteLine("Ha nacido un animal");
    }
}

// -----

public class Perro: Animal
{
```

```

public Perro()
{
    Console.WriteLine("Ha nacido un perro");
}
}

```

y su salida en pantalla, que recuerda a la del apartado 6.8, sería

```

Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un perro

```

Ejercicio propuesto:

(7.2.1) Crea una versión ampliada del ejercicio 6.8.1 (clase Trabajador y relacionadas), en la que no se cree un único objeto de cada clase, sino un array de tres objetos.

(7.2.2) Amplía el proyecto Libro (ejercicio 6.7.2), de modo que permita guardar hasta 1.000 libros. Main mostrará un menú que permita añadir un nuevo libro o ver los datos de los ya existentes.

(7.2.3) Amplía el esqueleto del ConsoleInvaders (6.7.3), para que haya 10 enemigos en una misma fila (todos compartirán una misma coordenada Y, pero tendrán distinta coordenada X). Necesitarás un nuevo constructor en la clase Enemigo, que reciba los parámetros X e Y.

Además, existe una peculiaridad curiosa: podemos crear un array de "Animales", que realmente **contenga objetos de distintas subclases** (en este caso, unos de ellos serán 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_07_02b.cs
```

```
// Array de objetos de distintas clases
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_02b
{
    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();
    }
}

// -----

public class Animal
{
    public Animal()
    {
        Console.WriteLine();
        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");
    }
}

```

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
Ha nacido un perro

Ha nacido un animal
Ha nacido un perro

Ha nacido un animal
Ha nacido un perro

Ha nacido un animal

```

Si los objetos tienen otros métodos, no sólo el constructor, no todo funcionará a la primera. Pronto veremos los posibles problemas y la forma de solucionarlos.

Ejercicios propuestos:

(7.2.4) A partir del ejemplo 07.02b y del ejercicio 6.8.1 (clase Trabajador y relacionadas), crea un array de trabajadores en el que no sean todos de la misma clase.

(7.2.5) Amplía el proyecto Libro (ejercicio 7.2.2), de modo que permita guardar 1000 documentos de cualquier tipo. A la hora de añadir un documento, se preguntará al usuario si desea guardar un documento genérico o un libro, para usar el constructor adecuado.

(7.2.6) Amplía el esqueleto del ConsoleInvaders (7.2.3), para que haya tres tipos de enemigos, y un array que contenga 3x10 enemigos (3 filas, cada una con 10 enemigos de un mismo tipo, pero distinto del tipo de los elementos de las otras filas). Cada tipo de enemigos será una subclase de Enemigo, que se distinguirá por usar una "imagen" diferente. Puedes usar la "imagen" que quieras (siempre que sea un string de letras, como "X" o "XX"). Si estas imágenes no se muestran

correctamente en pantalla al lanzar una partida, no te preocupes, lo solucionaremos en el siguiente apartado.

7.3. Funciones virtuales. La palabra "override"

En el ejemplo anterior hemos visto cómo crear un array de objetos, usando sólo la clase base en el momento de definirlo, pero insertando realmente objetos de cada una de las clases derivadas, y hemos visto que los constructores se llaman correctamente... pero con los métodos podemos encontrar problemas.

Vamos a verlo con un ejemplo que, en vez de tener constructores, va a tener un único método "Hablar", que se redefinirá en cada una de las clases hijas, y después comentaremos qué ocurre al ejecutarlo y cómo solucionarlo:

```
// Ejemplo_07_03a.cs
// Array de objetos, sin "virtual"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_03a
{
    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();
    }
}

// -----

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");
    }
}

```

(Recuerda que, como vimos en el apartado 6.4, ese **"new"** que aparece en "new void Hablar" sirve simplemente para anular un "warning" del compilador, que dice algo como "estás redifiniendo este método; añade la palabra new para indicar que eso es lo que deseas". Equivale a un "sí, sé que estoy redefiniendo este método").

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, el segundo es un gato y el tercero es un animal genérico:

```
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", y el array es de "animales", cada elemento hace lo que corresponde a un Animal genérico (dice "Estoy comunicándome"), a pesar de que hayamos indicado que se trata de un Perro u otra subclase distinta.

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 **pudiera contener distintos tipos** de animales.

Para conseguir este comportamiento, debemos indicar a nuestro compilador que el método "Hablar" que se usa en la clase Animal quizá sea **redefinido** por otras clases hijas, y que, en ese caso, debe prevalecer lo que indiquen las clases hijas.

La forma de conseguirlo es declarar ese método "Hablar" como "**virtual**" (para indicar al compilador que ese método probablemente será redefinido en las subclases), y emplear en las clases hijas la palabra "override" en vez de "new" (para dejar claro que debe reemplazar al método "virtual" original), así:

```
// Ejemplo_07_03b.cs
// Array de objetos, con "virtual"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_03b
{
    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();
    }
}

// -----

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");
    }
}

```

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...

Nota: Esto que nos acaba de ocurrir va a ser frecuente. Si el compilador nos avisa de que deberíamos añadir la palabra "new" porque estamos redefiniendo algo... es habitual que realmente lo que necesitamos sea "virtual" en la clase base y "override" en las clases derivada, especialmente si no se trata de un programa trivial, sino que vamos a tener objetos de varias clases coexistiendo en el programa, y especialmente si todos esos objetos van a ser parte de un único array o estructura similar (como las "listas", que veremos más adelante).

Ejercicios propuestos:

(7.3.1) Crea una versión ampliada del ejercicio 6.5.1 (Persona, PersonalInglesa, etc), en la que se cree un único array que contenga personas de varios tipos.

(7.3.2) Crea una variante del ejercicio 7.2.2 (array de Trabajador y derivadas), en la que se cree un único array "de trabajadores", que contenga un objeto de cada clase, y exista un método "Saludar" que se redefina en todas las clases hijas, usando "new" y probándolo desde "Main".

(7.3.3) Crea una variante del ejercicio anterior (7.3.2), que use "override" en vez de "new".

(7.3.4) Amplía el proyecto Libro (ejercicio 7.2.5): tanto la clase Documento como la clase Libro, tendrán un método ToString, que devuelva una cadena de texto formada por título, autor y ubicación, separados por guiones. Crea una clase Artículo, que añada el campo "procedencia". El cuerpo del programa permitirá añadir Artículos o Libros, no documentos genéricos. El método ToString deberá mostrar también el número de páginas de un libro y la procedencia de un artículo. La opción de mostrar datos llamará a los correspondientes métodos ToString. Recuerda usar "virtual" y "override" si en un primer momento no se comporta como debe.

(7.3.5) Amplía el esqueleto de ConsoleInvaders (7.2.6) para que muestre las imágenes correctas de los enemigos, usando "virtual" y "override". Además, cada tipo de enemigos debe ser de un color distinto. (Nota: para cambiar colores puedes usar `Console.ForegroundColor = ConsoleColor.Green;`). La nave que maneja el usuario debe ser blanca.

7.4. 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 de la superclase, 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_07_04a.cs
// Ejemplo de clases: Llamar a la superclase
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_04a
{
    public static void Main()
    {
        Gato miGato = new Gato();
        GatoSiames miGato2 = new GatoSiames();

        miGato.Hablar();
        Console.WriteLine(); // Línea en blanco
        miGato2.Hablar();
    }
}

// -----

public class Animal
{
    // Todos los detalles están en las subclases
}

// -----

public class Gato: Animal
{
    public void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}

// -----

public class GatoSiames: Gato
```

```

{
    public new void Hablar()
    {
        base.Hablar();
        Console.WriteLine("Pfff");
    }
}

```

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).

Ejercicios propuestos:

(7.4.1) Crea una versión ampliada del ejercicio 7.3.3, en la que el método "Hablar" de todas las clases hijas se apoye en el de la clase "Trabajador".

(7.4.2) Crea una versión ampliada del ejercicio 7.4.1, en la que el constructor de todas las clases hijas se apoye en el de la clase "Trabajador".

(7.4.3) Refina el proyecto Libro (ejercicio 7.3.4), para que el método ToString de la clase Libro se apoye en el de la clase Documento, y también lo haga el de la clase Artículo.

(7.4.4) Amplía el esqueleto de ConsoleInvaders (7.3.5) para que en Enemigo haya un único constructor que reciba las coordenadas X e iniciales. Los constructores de los tres tipos de enemigos deben basarse en éste.

7.5. La palabra "this": el objeto actual

Podemos hacer referencia al objeto que estamos usando, con "this". Un primer uso es dar valor a los atributos, incluso si los parámetros tienen el mismo nombre que éstos:

```
public void MoverA (int x, int y)
{
    this.x = x;
    this.y = y;
}
```

Un fuente completo sería así:

```
// Ejemplo_07_05a.cs
// Primer ejemplo de uso de "this": parámetros
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_05a
{
    public static void Main()
    {
        Titulo t = new Titulo(38,5,"Hola");
        t.Escribir();
    }
}

// -----

public class Titulo
{
    private int x;
    private int y;
    private string texto;

    public Titulo(int x, int y, string texto)
    {
        this.x = x;
        this.y = y;
        this.texto = texto;
    }

    public void Escribir()
    {
        Console.SetCursorPosition(x,y);
        Console.WriteLine(texto);
    }
}
```

En muchos casos, podemos evitar este uso de "this". 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 caso 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
}
```

En ese ejemplo, existiría un constructor de RectanguloRelleno que recibiera sólo la coordenada X (y que podría prefijar la Y al valor 0, por ejemplo), y también existe este otro constructor, que recibe X e Y, y que se basa en el anterior para fijar el valor de X.

Nuevamente, podríamos hacer un fuente completo de ejemplo:

```
// Ejemplo_07_05b.cs
// Segundo ejemplo de uso de "this": constructores
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_05b
{
    public static void Main()
    {
        Titulo t = new Titulo(38,5,"Hola");
        t.Escribir();
    }
}

// -----

public class Titulo
{
```

```

private int x;
private int y;
private string texto;

public Titulo(int nuevaX, int nuevaY, string txt)
{
    x = nuevaX;
    y = nuevaY;
    texto = txt;
}

public Titulo(int nuevaY, string txt)
    : this (40-txt.Length/2, nuevaY, txt)
{
}

public void Escribir()
{
    Console.SetCursorPosition(x,y);
    Console.WriteLine(texto);
}
}

```

La palabra "this" se usa mucho también para que **unos objetos "conozcan" a los otros**. Por ejemplo, en un juego de 2 jugadores, podríamos tener una clase Jugador, heredar de ella las clases Jugador1 y Jugador2, que serán muy parecidas entre sí, y nos podríamos sentir tentados de hacer que la clase Jugador tenga un "adversario" como atributo, y que el Jugador1 indique que su adversario es de tipo Jugador2, y lo contrario para el otro jugador, así:

```

// Ejemplo_07_05c.cs
// Dos clases que se usan mutuamente: recursividad indirecta
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_05c
{
    public static void Main()
    {
        Jugador1 j1 = new Jugador1();
        Jugador2 j2 = new Jugador2();
    }
}

// -----

public class Jugador
{
    protected Jugador adversario;
}

// -----

public class Jugador1 : Jugador

```

```

{
    public Jugador1()
    {
        adversario = new Jugador2();
    }
}

// -----

public class Jugador2 : Jugador
{
    public Jugador2()
    {
        adversario = new Jugador1();
    }
}

```

Pero este programa se queda colgado un instante, hasta terminar finalmente lanzando una **excepción de desbordamiento de pila** (Stack Overflow), porque estamos creando una recursividad indirecta sin fin: el jugador1 crea un jugador2, éste crea un nuevo jugador1 (en vez de utilizar el ya existente), que a su vez crea un nuevo jugador2 y así sucesivamente, hasta finalmente desbordar la zona de memoria que está reservada para llamadas recursivas.

Una alternativa mucho menos peligrosa (pero que, a cambio, complica el programa principal), es que sea el programa principal el que indique a cada jugador quién es su adversario:

```

// Ejemplo_07_05d.cs
// Dos clases que se usan mutuamente: "Main" coordina
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_05d
{
    public static void Main()
    {
        Jugador j1 = new Jugador();
        Jugador j2 = new Jugador();
        j1.SetAdversario(j2);
        j2.SetAdversario(j1);
    }
}

// -----

public class Jugador
{
    protected Jugador adversario;

    public void SetAdversario(Jugador nuevoAdversario)
    {
        adversario = nuevoAdversario;
    }
}

```

```
    }
}
```

Otra alternativa es que un Jugador le pueda decir a otro "yo soy tu adversario", y ese "yo" equivaldrá a un "this". En general, eso simplificará el programa principal, a cambio de complicar ligeramente las clases auxiliares:

```
// Ejemplo_07_05e.cs
// Dos clases que se usan mutuamente: "this"
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_07_05e
{
    public static void Main()
    {
        Jugador j1 = new Jugador(); // Creado sin más detalles
        Jugador j2 = new Jugador(j1); // Indicando su adversario
    }
}

// -----

public class Jugador
{
    protected Jugador adversario;

    public Jugador()
    {
    }

    public Jugador(Jugador adversario)
    {
        // Mi adversario es el que me indican
        SetAdversario( adversario );

        // Y yo soy su adversario
        adversario.SetAdversario( this );
    }

    public void SetAdversario(Jugador nuevoAdversario)
    {
        adversario = nuevoAdversario;
    }
}
```

Hay que recordar que, en general, **cundo una clase contiene a otras**, la clase contenedora sabe los detalles de las clases contenidas (la "casa" conoce a sus "puertas"), pero las clases contenidas no saben nada de la clase que las contiene (las "puertas" no saben otros detalles de la "casa" a la que pertenecen). Si queremos que se puedan comunicar con la clase contenedora, deberemos usar

"this" para que la conozcan, en vez de crear una nueva clase contenedora con "new", o provocaremos una recursividad indirecta sin fin, como en el primer ejemplo.

Ejercicios propuestos:

(7.5.1) Crea una versión ampliada del ejercicio 7.4.2, en la que el constructor sin parámetros de la clase "Trabajador" se apoye en otro constructor que reciba como parámetro el nombre de esa persona. La versión sin parámetros asignará el valor "Nombre no detallado" al nombre de esa persona.

(7.5.2) Crea una clase Puerta con un ancho, un alto y un método "MostrarEstado" que muestre su ancho y su alto. Crea una clase Casa, que contenga 3 puertas y otro método "MostrarEstado" que escriba "Casa" y luego muestre el estado de sus tres puertas.

(7.5.3) Crea una clase Casa, con una superficie (por ejemplo, 90 m²) y un método "MostrarEstado" que escriba su superficie. Cada casa debe contener 3 puertas. Las puertas tendrán un ancho, un alto y un método "MostrarEstado" que muestre su ancho y su alto y la superficie de la casa en la que se encuentran. Crea un programa de prueba que cree una casa y muestre sus datos y los de sus tres puertas.

(7.5.4) Amplía el esqueleto de ConsoleInvaders (7.4.4), de modo que el constructor sin parámetros de la clase Nave se apoye en el constructor con parámetros de la misma clase, prefijando unas coordenadas que te parezcan las más adecuadas.

7.6. 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. Así, para sumar dos matrices, en vez de hacer algo como "matriz3 = suma(matriz1, matriz2)" podríamos hacer simplemente "matriz3 = matriz1 + matriz2"

No entraremos mucho más en detalle, pero la idea es que redefiniríamos un método llamado "operador +", y que devolvería un dato estático 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 < tamanyo; x++)
        for (int y=0; y < tamanyo; 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;
```

Eso sí, debes tener presente que "no todo se puede redefinir". Por ejemplo, puedes pensar en sobrecargar los operadores [y] para acceder a un cierto elemento de la matriz, pero C# no permite redefinir cualquier operador, y los corchetes están entre los que no podremos cambiar. En su lugar, deberías crear métodos Get para acceder a una posición y métodos Set para cambiar el valor de una posición.

Ejercicios propuestos:

(7.6.1) 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. (Nota: en C# puedes sobrecargar el operador "+", pero no el operador "[]", de modo que tendrás que crear métodos "get" y "set" para leer los valores de posiciones de la matriz y para cambiar su valor).

(7.6.2) Si has estudiado los "números complejos", crea una clase "Complejo", que represente a un número complejo (formado por una parte "real" y una parte "imaginaria"). Incluye un constructor que reciba ambos datos como parámetros. Crea también métodos "get" y "set" para leer y modificar los valores de dichos datos, así como métodos que permitan saber los valores de su módulo y su argumento. Crea un método que permita sumar un complejo a otro, y redefine el operador "+" como forma alternativa de realizar esa operación.

(7.6.3) Crea una clase "Fraccion", que represente a una fracción, formada por un numerador y un denominador. Crea un constructor que reciba ambos datos como parámetros y otro constructor que reciba sólo el numerador. Crea un método

Escribir, que escriba la fracción en la forma "3 / 2". Redefine los operadores "+", "-", "*" y "/" para que permitan realizar las operaciones habituales con fracciones.

(7.6.4) Crea una clase "Vector3", que represente a un vector en el espacio de 3 dimensiones. Redefine los operadores "+" y "-" para sumar y restar dos vectores, "*" para hallar el producto escalar de dos vectores y "%" para calcular su producto vectorial.

7.7. Proyectos completos propuestos

La mejor forma de aplicar todos los conocimientos sobre clases y de ponerlos a prueba es crear algún proyecto de una cierta complejidad, que incluya diferentes clases, herencia, arrays de objetos, comunicación entre dichos objetos, etc. Aquí tienes unos cuantos ejercicios propuestos, sin solución (y para los que no existirá una solución única):

(7.7.1) Crea un proyecto "Space Invaders" con todas las clases que vimos al principio del tema 6. El proyecto no será jugable, pero deberá compilar correctamente.

(7.7.2) Crea un proyecto "PersonajeMovil", que será un esqueleto sobre el que se podría ampliar para crear un juego de plataformas. Existirá una pantalla de bienvenida, una pantalla de créditos y una partida, en la que el usuario podrá mover a un personaje (que puede ser representado por una letra X). Las teclas de movimiento quedan a tu criterio, pero podrían ser "wasd" o las flechas del cursor. Si quieres que el movimiento sea más suave, puedes investigar sobre Console.ReadKey (que posiblemente habrás usado ya en el proyecto ConsoleInvaders) como alternativa a Console.ReadLine.

(7.7.3) Crea un proyecto "Laberinto", a partir del proyecto "PersonajeMovil", añadiendo tres enemigos que se muevan de lado a lado de forma autónoma y un laberinto de fondo, cuyas paredes no se puedan "pisar" (la clase Laberinto puede tener métodos que informen sobre si el jugador podría moverse a ciertas coordenadas X e Y). Si el personaje toca a un enemigo, acabará la partida y se regresará a la pantalla de bienvenida.

(7.7.4) Crea un proyecto "Laberintos", a partir del proyecto "Laberinto", añadiendo premios que recoger, tres vidas para nuestro personaje y varios laberintos distintos que recorrer.

(7.7.5) Crea un proyecto "SistemaDomotico", que simule un sistema domótico para automatizar ciertas funciones en una casa: apertura y cierre de ventanas y puertas, encendido de calefacción, etc. Además de esos elementos físicos de la casa, también existirá un panel de control, desde el que el usuario podría controlar el resto de elementos, así como programar el comportamiento del sistema (por

ejemplo, subir una persiana inmediatamente o hacer que la calefacción se encienda todos los días desde las 19h hasta las 23h, a una cierta temperatura).

(7.7.6) Completa el proyecto Libro (ejercicio 7.4.3), para que las clases de datos tengan un método "Contiene", que reciba un texto y devuelva el valor booleano "true" cuando ese texto esté contenido en el título o en el autor (y en la procedencia, para los artículos). El Main deberá permitir al usuario realizar búsquedas, aprovechando esta nueva funcionalidad.

(7.7.7) Añade al proyecto Libro cualquier otra funcionalidad que te parezca interesante, como la de modificar datos, borrarlos u ordenarlos.

(7.7.8) Crea en el proyecto Libro una clase ListaDeDocumentos, que oculte a Main los detalles de que internamente se trata de un array: deberá permitir opciones como añadir un documento, obtener los datos de uno de ellos, buscar entre la lista de todos ellos, etc. Los criterios de diseño quedan a tu criterio. Por ejemplo, puedes hacer que la búsqueda devuelve un array con los números de ficha encontrados, o un array con el contenido de esas fichas, o el contenido la siguiente ficha que cumpla con esos criterios.

(7.7.9) Amplía el esqueleto de ConsoleInvaders (7.5.4), con una clase "BloqueDeEnemigos", que será la que contenga el array de enemigos, de modo que se simplifique la lógica de la clase Partida. Esta clase tendrá un método Dibujar, que mostrará todo el array en pantalla, y un método Mover, que moverá todo el bloque hacia la derecha y la izquierda de forma alternativa (deberás comprobar la posición inicial del primer enemigo y la posición final del último enemigo). En cada pasada por el bucle de juego deberás llamar a Mover.

(7.7.10) Crea una clase Disparo en ConsoleInvaders. Cuando el usuario pulse cierta tecla (Espacio, por ejemplo), aparecerá un disparo encima de la nave, y se moverá hacia arriba hasta que desaparezca por la parte superior de la pantalla. Existirá un único disparo, y no se podrá volver a disparar si está activo (en pantalla). Inicialmente estará desactivado, y lo volverá a estar cuando llegue al margen de la pantalla.

(7.7.11) En ConsoleInvaders, crea un método "ColisionaCon" en la clase Sprite, que reciba otro Sprite como parámetro y devuelva el valor booleano "true" si ambos sprites están "chocando" o "false" en caso contrario. Tendrás que pensar qué relación habrá entre las coordenadas X e Y de ambos sprites para que "choquen".

(7.7.12) En ConsoleInvaders, crea una clase Ovni, con un nuevo tipo de Enemigo que no estará siempre activo. Su método Mover hará que se mueva hacia la derecha si ya está activo o, en caso contrario, que genere un número al azar para decidir si debe activarse.

(7.7.13) En ConsoleInvaders, comprueba colisiones entre el disparo y el Ovni. Si hay colisión, desaparecerán ambos y el jugador obtendrá 50 puntos.

(7.7.14) En ConsoleInvaders, comprueba colisiones entre el disparo y el bloque de enemigos. Si el disparo toca algún enemigo, ambos desaparecerán y el jugador obtendrá 10 puntos.

(7.7.15) En ConsoleInvaders, añade una clase "Marcador", que muestre la puntuación y la cantidad de vidas restantes (que por ahora, siempre será 3).

(7.7.16) En ConsoleInvaders, añade la posibilidad de que algunos enemigos al azar puedan disparar (los disparos de los enemigos "va hacia abajo"; piensa si crear una nueva clase o ampliar las funcionalidades de la clase Disparo que ya tienes). Ajusta la frecuencia de disparos, de modo que el juego continúe siendo jugable. Si uno de esos disparos impacta con la nave, se perderá una vida y el disparo desaparecerá. Si se pierden las 3 vidas, acaba la partida y se volverá a la pantalla de presentación.

(7.7.17) En ConsoleInvaders, crea la estructura que sea necesaria para almacenar las "mejores puntuaciones", que se actualizarán al terminar cada partida y se mostrarán en la pantalla de bienvenida.

(7.7.18) En ConsoleInvaders, añade las "torres defensivas", que protegen al jugador y que se van rompiendo poco a poco cada vez que un disparo impacta con ellas.

8. Manejo de ficheros

8.1. Escritura en un fichero de texto

Cuando queramos manipular un fichero, siempre deberemos hacerlo **en tres pasos**:

- Abrir el fichero.
- Leer datos de él o escribir datos en él.
- Cerrar el fichero.

Eso sí, no siempre conseguiremos 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 tratemos de 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_08_01a.cs
// Escritura en un fichero de texto
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;    // Para StreamWriter

public class Ejemplo_08_01a
{
    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 de texto en el que podemos escribir información.
- El fichero de texto lo creamos con el método (estático) CreateText, que pertenece a la clase File.

- Para escribir en el fichero, empleamos Write y WriteLine, al igual que hacemos en la consola.
- Finalmente, debemos cerrar el fichero con Close, o podría quedar algún dato sin guardar.

Una forma alternativa de crear el fichero es no usar "File.CreateText", sino un constructor que reciba el nombre de fichero, así:

```
// Ejemplo_08_01b.cs
// Escritura en un fichero de texto, con constructor
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;    // Para StreamWriter

public class Ejemplo_08_01b
{
    public static void Main()
    {
        StreamWriter fichero = new StreamWriter("prueba.txt");
        fichero.WriteLine("Esto es una línea");
        fichero.Write("Esto es otra");
        fichero.WriteLine(" y esto es continuación de la anterior");
        fichero.Close();
    }
}
```

Existe otra sintaxis, más compacta, que nos permite olvidarnos (hasta cierto punto) de cerrar el fichero, empleando la orden "using" para delimitar el bloque que lee datos del fichero, de esta forma:

```
// Ejemplo_08_01c.cs
// Escritura en un fichero de texto, con "using"
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;    // Para StreamWriter

public class Ejemplo_08_01c
{
    public static void Main()
    {
        using (StreamWriter fichero = new StreamWriter("prueba.txt"))
        {
            fichero.WriteLine("Esto es una línea");
            fichero.Write("Esto es otra");
            fichero.WriteLine(" y esto es continuación de la anterior");
        }
    }
}
```

Ejercicios propuestos:

(8.1.1) 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).

(8.1.2) Crea una versión de la base de datos de ficheros (ejercicio 5.2.3), de modo que todos los datos se vuelquen a un fichero de texto al terminar la ejecución del programa.

(8.1.3) Amplia el proyecto Libro (ejercicio 7.7.8), de modo que todos los datos se vuelquen a un fichero de texto al terminar la ejecución del programa.

8.2. Lectura de un fichero de texto

La estructura de un programa que leyera de un fichero de texto sería parecida a:

```
// Ejemplo_08_02a.cs
// Lectura de un fichero de texto
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;    // Para StreamReader

public class Ejemplo_08_02a
{
    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.

Nuevamente, podemos usar un constructor en vez de OpenText:

```
// Ejemplo_08_02b.cs
// Lectura de un fichero de texto, con constructor
```

```
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;    // Para StreamReader

public class Ejemplo_08_02b
{
    public static void Main()
    {
        StreamReader fichero = new StreamReader("prueba.txt");
        string linea = fichero.ReadLine();
        Console.WriteLine( linea );
        Console.WriteLine( fichero.ReadLine() );
        fichero.Close();
    }
}
```

o bien utilizar la sintaxis alternativa, con la palabra "using":

```
// Ejemplo_08_02c.cs
// Lectura de un fichero de texto, con "using"
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;    // Para StreamReader

public class Ejemplo_08_02c
{
    public static void Main()
    {
        using (StreamReader fichero = new StreamReader("prueba.txt"))
        {
            string linea = fichero.ReadLine();
            Console.WriteLine( linea );
            Console.WriteLine( fichero.ReadLine() );
        }
    }
}
```

Ejercicios propuestos:

- (8.2.1)** Crea un programa que lea las tres primeras líneas del fichero creado en el ejercicio 8.1.1 y las muestre en pantalla. Usa `OpenText` para abrirlo.
- (8.2.2)** Crea una versión alternativa del ejercicio 8.2.1, usando el constructor de `StreamReader`.
- (8.2.3)** Crea una versión alternativa del ejercicio 8.2.2, empleando la sintaxis alternativa de "using".

8.3. Lectura hasta el final del fichero

Normalmente no queremos 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.

Habitualmente, si queremos procesar todo un fichero, esta lectura y comprobación debería ser repetitiva, así:

```
// Ejemplo_08_03a.cs
// Lectura de un fichero de texto completo
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;
public class Ejemplo_08_03a
{
    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();
    }
}
```

Por supuesto, podrías emplear el constructor o "using" para acceder al fichero. Esos cambios serán parte de los ejercicios propuestos.

Ejercicios propuestos:

(8.3.1) Crea una variante del ejemplo 08_03a, empleando un constructor en vez de "File.OpenText".

(8.3.2) Crea una variante del ejemplo 08_03a, empleando "using" en vez de "Close".

(8.3.3) Prepara un programa que pregunte un nombre de fichero y muestre en pantalla el contenido de ese fichero, haciendo una pausa después de cada 24 líneas, para que dé tiempo a leerlo. Cuando el usuario pulse la tecla Intro, se mostrarán las siguientes 24 líneas, y así sucesivamente hasta que termine el fichero.

(8.3.4) Amplía la base de datos de ficheros (ejercicio 8.1.2), de modo que los datos se lean desde fichero (si existe) en el momento de lanzar el programa (puedes usar try-catch para que el programa no falle en el momento inicial, en el que quizá

todavía no existan datos en fichero; dentro de poco veremos formas alternativas de saber si existe).

(8.3.5) Amplia el proyecto Libro (ejercicio 8.1.3), de modo que los datos se lean desde fichero (si existe) en el momento de poner el programa en marcha.

(8.3.6) Crea 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 líneas 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).

8.4. 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_08_04a.cs
// Añadir a un fichero de texto
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_04a
{
    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();
    }
}
```

También podemos usar un constructor para añadir datos a un fichero existente, pero entonces tendremos que emplear un segundo parámetro booleano, que será "true" para añadir (o "false" para sobrescribir el fichero, si ya existiera):

```
// Ejemplo_08_04b.cs
// Añadir a un fichero de texto, con constructor y "using"
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;
```

```

public class Ejemplo_08_04b
{
    public static void Main()
    {
        using (StreamWriter fichero = new StreamWriter("prueba2.txt", false))
        {
            fichero.WriteLine("Primera línea");
        }

        using (StreamWriter fichero = new StreamWriter("prueba2.txt", true))
        {
            fichero.WriteLine("Segunda línea");
        }
    }
}

```

Ejercicios propuestos:

(8.4.1) Un programa que pida al usuario que teclee frases, y las almacene en el fichero "log.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.

8.5. Ficheros en otras carpetas

Si un fichero al que queremos acceder se encuentra en otra carpeta, basta con que el nombre de fichero incluya también la ruta.

Debemos recordar 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:

(8.5.1) Crea 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 subdirectorío llamado "resultados". Cada vez

que se ejecute el programa, deberá añadir los nuevos resultados a continuación de los resultados de las ejecuciones anteriores.

8.6. 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 o de que introduzcamos un nombre incorrecto.

Una primera solución es usar "File.Exists(nombre)", para comprobar si está, antes de intentar abrirlo:

```
// Ejemplo_08_06a.cs
// Saber si un fichero existe
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_06a
{
    public static void Main()
    {
        StreamReader fichero;
        string nombre;

        while (true) // Interrumpimos desde dentro con "break"
        {
            Console.Write( "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 línea es: {0}",
                    fichero.ReadLine() );
                fichero.Close();
            }
            else
                Console.WriteLine( "No existe!" );
        }
    }
}
```

Ejercicios propuestos:

(8.6.1) Mejora el ejercicio 8.3.4 para que compruebe antes si el fichero existe, y muestre un mensaje de aviso en caso de que no sea así.

(8.6.2) Mejora el ejemplo 08_06a para que no use "while (true)", sino una variable booleana de control.

Otra forma de comprobar si un fichero existe o no es usar "excepciones", con las que ya nos habíamos encontrado en el tema 3 y que veremos con más detalle en el próximo apartado.

8.7. 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).

Por ello, una forma más eficaz de comprobar si ha existido algún tipo de error es comprobar las posibles "excepciones", con las que ya tuvimos un contacto al final del tema 2.

Típicamente, los pasos que puedan ser problemáticos irán dentro del bloque "try" y los mensajes de error y/o acciones correctoras estarán en el bloque "catch". Así, 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_08_07a.cs
// Excepciones y ficheros (1)
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_07a
{
    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);
        }
    }
}
```

```

    }
    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 algo como

Introduzca el nombre del fichero

prueba

Ha habido un error: No se pudo encontrar el archivo
'C:\Fuentes\nacho\prueba'.

Pero, como ya vimos, generalmente 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 que una simple "Exception", de forma que el mensaje de aviso sea más concreto, o que podamos dar pasos más adecuados para solucionar el problema:

```

// Ejemplo_08_07b.cs
// Excepciones y ficheros (2)
// Introducción a C#, por Nacho Cabanes

```

```

using System;
using System.IO;

public class Ejemplo_08_07b
{
    public static void Main()
    {
        StreamWriter fichero;
        string nombre;
    }
}

```

```

string linea;

Console.Write("Introduzca el nombre del fichero: ");
nombre = Console.ReadLine();
Console.Write("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);
}
}
}

```

Las excepciones más generales (IOException y, sobre todo, Exception) deberán ser las últimas que analicemos, y las más específicas deberán ser las primeras.

Hay que recordar que, 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 esperamos que introduzca un número, pero, en lugar de ello, tecleado un texto).

Ejercicios propuestos:

(8.7.1) 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.

(8.7.2) 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.

(8.7.3) 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.

8.8. Conceptos básicos sobre ficheros

Llega el momento de concretar 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 cassette). 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, en los ficheros binarios (en un fichero de texto en general no podremos hacerlo, porque no sabremos el tamaño de cada línea de texto, así que no podremos saltar hasta una cierta línea).

8.9. Leer un byte 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 fichero de sonido MP3.

Como primer acercamiento, vamos a ver cómo abrir un fichero (no necesariamente de texto) y leer el primer byte que contiene. Comenzaremos por utilizar la clase **FileStream**, diseñada para acceder a un byte o a bloques de bytes. Esta clase tiene un método **ReadByte**, que devuelve un entero con valor -1 en caso de error, o un dato que podríamos convertir a byte si la lectura ha sido correcta, así:

```
// Ejemplo_08_09a.cs
// Ficheros binarios: lectura de un byte con FileStream
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_09a
{
    public static void Main()
    {
        FileStream fichero;
        string nombre;
        byte unDato;

        Console.WriteLine("Introduzca el nombre del fichero");
        nombre = Console.ReadLine();

        try
        {
            fichero = File.OpenRead(nombre);
            unDato = (byte) fichero.ReadByte();
            Console.WriteLine("El byte leído es un {0}",
                             unDato);
            fichero.Close();
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: " + e.Message);
            return;
        }
    }
}
```

Podemos usar también un **constructor** para acceder al fichero. Este constructor recibirá dos parámetros: el nombre del fichero y el modo de apertura. Por ahora, nos interesará un único modo, que será "FileMode.Open", que nos permite leer datos desde un fichero que ya existe:

```
// Ejemplo_08_09b.cs
// Ficheros binarios: lectura de un byte con FileStream, constructor
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_09b
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
        string nombre = Console.ReadLine();

        try
        {
            FileStream fichero = new FileStream(nombre, FileMode.Open);
            byte unDato = (byte) fichero.ReadByte();
            Console.WriteLine("El byte leído es un {0}",
                unDato);
            fichero.Close();
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: "+e.Message);
            return;
        }
    }
}
```

Y también podemos emplear la sintaxis alternativa, con "**using**", de modo que no sea necesario cerrar el fichero de forma explícita:

```
// Ejemplo_08_09c.cs
// Ficheros binarios: lectura de un byte con FileStream, using
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_09c
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
        string nombre = Console.ReadLine();

        try
        {
            using (FileStream fichero = new FileStream(
                nombre, FileMode.Open))
            {
                byte unDato = (byte) fichero.ReadByte();
                Console.WriteLine("El byte leído es un {0}",
                    unDato);
            }
        }
        catch (Exception e)
        {
        }
    }
}
```

```

    {
        Console.WriteLine("Error: "+e.Message);
        return;
    }
}

```

Los otros **modos de apertura** posibles (cuyos nombre en inglés son casi autoexplicativos) son: **Append** (añade al final o crea si no existe), **Create** (crea el fichero o lo sobrescribe si ya existía), **CreateNew** (crea si no existía, o falla si existe), **Open** (abre si existe o falla si no existe), **OpenOrCreate** (abre si existe o lo crea si no existe), **Truncate** (abre un fichero que debe existir y lo vacía).

Si quisiéramos **leer un segundo dato**, volveríamos a llamar a `ReadByte`, porque tras cada lectura, avanza la posición actual dentro del fichero.

Ejercicios propuestos:

(8.9.1) Abre un fichero con extensión EXE (cuyo nombre introducirá el usuario) y comprueba si realmente se trata de un ejecutable, mirando si los dos primeros bytes del fichero son un 77 (que corresponde a una letra "M", según el estándar que marca el código ASCII) y un 90 (una letra "Z"), respectivamente.

(8.9.2) Abre una imagen BMP (cuyo nombre introducirá el usuario) y comprueba si realmente se trata de un fichero en ese formato, mirando si los dos primeros bytes del fichero corresponden a una letra "B" y una letra "M", respectivamente.

(8.9.3) Abre una imagen GIF y comprueba si sus tres primeros bytes son las letras G, I, F.

8.10. Leer hasta el final de un fichero binario

Tenemos dos formas de leer byte a byte todos datos que forma un `FileStream`. La primera es descubrir el tamaño del fichero, que podemos obtener con `".Length"`. A partir de entonces, podríamos usar un "for" para recorrer todo el fichero, así:

```

// Ejemplo_08_10a.cs
// Ficheros binarios: lectura completa de un FileStream
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_10a
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
    }
}

```



```

string nombre = Console.ReadLine();
FileStream fichero = File.OpenRead(nombre);
int count = 0;
for ( int pos = 0; pos < fichero.Length; pos++)
{
    byte unDato = (byte) fichero.ReadByte();
    char letra = Convert.ToChar( unDato );
    if (letra == 'A')
        count++;
}
Console.WriteLine("Tiene {0} letras A", count);
fichero.Close();
}
}

```

(Por simplicidad, este fuente no realiza ningún tipo de **comprobación de errores**. Lo mismo ocurrirá en la mayoría de fuentes de ejemplo de este tema, pero un programa real sí debería comprobar si el fichero existe y si se ha producido algún error durante la lectura).

La segunda forma de leer hasta el final del fichero es comprobar si el dato leído es un -1. Cuando esto ocurra, será que señal de que se ha llegado al final del fichero. El fuente correspondiente podría ser:

```

// Ejemplo_08_10b.cs
// Ficheros binarios: lectura completa de un FileStream
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_10b
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
        string nombre = Console.ReadLine();
        FileStream fichero = File.OpenRead(nombre);
        int count = 0;
        int dato;
        do
        {
            dato = fichero.ReadByte();
            if (dato != -1)
            {
                char letra = Convert.ToChar( (byte) dato );
                if (letra == 'A')
                    count++;
            }
        }
        while (dato != -1);
        Console.WriteLine("Tiene {0} letras A", count);
        fichero.Close();
    }
}

```

Ejercicios propuestos:

(8.10.1) Crea un programa que compruebe si un fichero (cuyo nombre introducirá el usuario) contiene una cierta letra (también escogida por el usuario).

(8.10.2) Crea un programa que cuente la cantidad de vocales que contiene un fichero binario.

(8.10.3) Crea un programa que diga si un fichero (binario) contiene una cierta palabra que introduzca el usuario.

(8.10.4) Crea un programa que extraiga a un fichero de texto todos los caracteres alfabéticos (códigos 32 a 127, además del 10 y el 13) que contenga un fichero binario.

8.11. Leer bloques de datos de un fichero binario

Leer byte a byte 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, dentro de la clase "FileStream" tenemos 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 (no la posición en el fichero, sino en el array, de modo que casi siempre será 0, es decir, al principio del array), 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 de lectura o si hemos llegado al final del fichero).

```
// Ejemplo_08_11a.cs
// Ficheros binarios: lectura por bloques de un FileStream
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_11a
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
        string nombre = Console.ReadLine();

        FileStream fichero = File.OpenRead(nombre);
        byte[] datos = new byte[10];
        int posicion = 0;
        int cantidadALeer = 10;
        int 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 leído es {0}", datos[0]);
            Console.WriteLine("El tercero es {0}", datos[2]);
        }
        fichero.Close();
    }
}

```

Ejercicios propuestos:

(8.11.1) Abre 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. Debes leer ambos bytes a la vez, usando un array.

(8.11.2) Abre una imagen BMP (cuyo nombre introducirá el usuario) y comprueba si realmente se trata de un fichero en ese formato, mirando si los dos primeros bytes del fichero corresponden a una letra "B" y una letra "M", respectivamente. Usa "Read" y un array.

(8.11.3) Abre una imagen GIF y comprueba si sus tres primeros bytes son las letras G, I, F. Guarda los 3 datos en un array.

(8.11.4) Abre una imagen en formato BMP y comprueba si está comprimida, mirando el valor del byte en la posición 30 (empezando a contar desde 0). Si ese valor es un 0 (que es lo habitual), indicará que el fichero no está comprimido. Deberás leer toda la cabecera (los primeros 54 bytes) con una sola orden.

8.12. La posición en el fichero

Cuando leemos byte a byte (y también a veces en el caso de leer todo un bloque), habitualmente nos interesará situarnos antes en la posición exacta en la que se encuentra el dato que nos interesa leer. En un fichero binario, esto podemos conseguirlo sin necesidad de leer todos los bytes anteriores, accediendo directamente a la posición que buscamos (acceso aleatorio).

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**" (y también sabemos que podemos obtener la longitud del fichero, mirando su propiedad "Length").

```
// Ejemplo_08_12a.cs
// Ficheros binarios: posición
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_12a
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
        string nombre = Console.ReadLine();
        FileStream fichero = File.OpenRead(nombre);
        byte[] datos = new byte[10];
        int posicion = 0;
        int cantidadALeer = 10;
        int 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 leído 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.Close();
    }
}
```

(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).

Ejercicios propuestos:

(8.12.1) Abre un fichero con extensión EXE y comprueba si su segundo byte corresponde a una letra "Z", sin leer su primer byte.

(8.12.2) Abre una imagen en formato BMP y comprueba si está comprimida, mirando el valor del byte en la posición 30 (empezando a contar desde 0). Si ese

valor es 0 (que es lo habitual), indicará que el fichero no está comprimido. Salta a esa posición directamente, sin leer toda la cabecera.

8.13. Leer datos nativos

Un FileStream es cómodo para leer byte a byte, pero no tanto cuando queremos leer otros tipos de datos básicos existentes en C# (short, int, float, etc.). Para eso usaremos la clase "BinaryReader". Por ejemplo, podríamos leer el entero corto que forman los dos bytes iniciales de un fichero con:

```
// Ejemplo_08_13a.cs
// Ficheros binarios: lectura de un short, con BinaryReader
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_13a
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
        string nombre = Console.ReadLine();

        BinaryReader fichero = new BinaryReader(
            File.Open(nombre, FileMode.Open));
        short dato = fichero.ReadInt16();
        Console.WriteLine("El dato leído es {0}", dato);
        fichero.Close();
    }
}
```

Si preferimos hacer esta lectura byte a byte, podremos usar tanto un FileStream como un BinaryReader, pero deberemos recomponer ese "short", teniendo en cuenta que, en la mayoría de sistemas actuales, en primer lugar aparece el byte menos significativo y luego el byte más significativo, de modo que el dato sería $\text{byte1} + \text{byte2} \times 256$, así:

```
// Ejemplo_08_13b.cs
// Ficheros binarios: lectura de un short, byte a byte
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_13b
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
        string nombre = Console.ReadLine();
```

```

        BinaryReader fichero = new BinaryReader(
            File.Open(nombre, FileMode.Open));
        short dato1 = fichero.ReadByte();
        short dato2 = fichero.ReadByte();
        Console.WriteLine("El dato leído es {0}", dato1 + dato2*256);
        fichero.Close();
    }
}

```

También podemos emplear la cláusula "using", para que el fichero se cierre automáticamente, así:

```

// Ejemplo_08_13c.cs
// Ficheros binarios: lectura de un short, con BinaryReader, using
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_13c
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero");
        string nombre = Console.ReadLine();

        using (BinaryReader fichero = new BinaryReader(
            File.Open(nombre, FileMode.Open)))
        {
            short dato = fichero.ReadInt16();
            Console.WriteLine("El dato leído es {0}", dato);
        }
    }
}

```

Ejercicios propuestos:

(8.13.1) Abre un fichero con extensión EXE y comprueba si comienza con el entero corto 23117.

(8.13.2) Abre una imagen en formato BMP y comprueba si comienza con el entero corto 19778.

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, a su "BaseStream", así: `ficheroEntrada.BaseStream.Seek(1, SeekOrigin.Begin);`

Por ejemplo, la anchura de una imagen en formato BMP es un entero de 32 bits que se encuentra en la posición 18, de modo que podríamos hacer:

```
// Ejemplo_08_13d.cs
// Ficheros binarios: Seek, con BinaryReader
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_13d
{
    public static void Main()
    {
        Console.WriteLine("Introduzca el nombre del fichero BMP");
        string nombre = Console.ReadLine();

        BinaryReader fichero = new BinaryReader(
            File.Open(nombre, FileMode.Open));
        fichero.BaseStream.Seek(18, SeekOrigin.Begin);
        int ancho = fichero.ReadInt32();
        Console.WriteLine("El ancho es {0}", ancho);
        fichero.Close();
    }
}
```

Ejercicios propuestos:

(8.13.3) El alto de un fichero BMP es un entero de 32 bits que se encuentra en la posición 22. Amplía el ejemplo 08_13d, para que muestre también el alto de ese fichero BMP.

8.14. Ejemplo completo: leer información de un fichero BMP

Hemos estado extrayendo algo de información de ficheros BMP. Ahora vamos a ir un poco más allá y a tratar de obtener muchos de los detalles que hay en su cabecera. El formato de dicha cabecera es el siguiente:

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

Por tanto, será fácil hacer que se nos muestren algunos detalles, como su ancho, su alto, la resolución y si la imagen está comprimida o no:

```
// Ejemplo_08_14a.cs
// Información de un fichero BMP, con BinaryReader
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_14a
{
    public static void Main()
    {
        Console.WriteLine("Información de imágenes BMP");
        Console.WriteLine("Dime el nombre del fichero: ");
        string nombre = Console.ReadLine();

        if (!File.Exists( nombre) )
        {
            Console.WriteLine("No encontrado!");
        }
        else
        {
            BinaryReader fichero = new BinaryReader(
                File.Open(nombre, FileMode.Open));

            // Leo los dos primeros bytes
            char marca1 = Convert.ToChar( fichero.ReadByte() );
            char marca2 = Convert.ToChar( fichero.ReadByte() );

            if ((marca1 != 'B') || (marca2 != 'M'))
                Console.WriteLine("No parece un fichero BMP");
            else
            {
                Console.WriteLine("Marca del fichero: {0}{1}",
                    marca1, marca2);
            }
        }
    }
}
```



```
// Posición 18: ancho
fichero.BaseStream.Seek(18, SeekOrigin.Begin);
int ancho = fichero.ReadInt32();
Console.WriteLine("Ancho: {0}", ancho);

// A continuación: alto
int alto = fichero.ReadInt32();
Console.WriteLine("Alto: {0}", alto);

// Posición 30: compresión
fichero.BaseStream.Seek(30, SeekOrigin.Begin);
int compresion = fichero.ReadInt32();
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;
}

// 4 bytes después: resolución horizontal
fichero.BaseStream.Seek(4, SeekOrigin.Current);
int resolH = fichero.ReadInt32();
Console.WriteLine("Resolución Horiz.: {0}", resolH);

// A continuación: resolución vertical
int resolV = fichero.ReadInt32();
Console.WriteLine("Resolución Vertical: {0}", resolV);

fichero.Close();
}
}
}
```

Podemos hacerlo también con un `FileStream`, que simplificará ciertas operaciones, como la lectura, que se puede hacer toda en un bloque, pero a cambio complicará otras operaciones, como el cálculo de los valores enteros, que habrá que componer a partir de los 4 bytes que los forman:

```
// Ejemplo_08_14b.cs
// Información de un fichero BMP, con FileStream
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_14b
{
    public static void Main()
    {
        Console.WriteLine("Información de imágenes BMP");
        Console.WriteLine("Dime el nombre del fichero: ");
        string nombre = Console.ReadLine();
```

```

if (! File.Exists( nombre ) )
{
    Console.WriteLine("No encontrado!");
}
else
{
    FileStream fichero = File.OpenRead(nombre);
    int tamanyoCabecera = 54;
    byte[] cabecera = new byte[tamanyoCabecera];

    int cantidadLeida = fichero.Read(cabecera, 0, tamanyoCabecera);
    fichero.Close();

    if (cantidadLeida != tamanyoCabecera)
    {
        Console.WriteLine("No se ha podido leer la cabecera");
    }
    else
    {
        // Analizo los dos primeros bytes
        char marca1 = Convert.ToChar( cabecera[0] );
        char marca2 = Convert.ToChar( cabecera[1] );

        if ((marca1 != 'B') || (marca2 != 'M'))
            Console.WriteLine("No parece un fichero BMP");
        else
        {
            Console.WriteLine("Marca del fichero: {0}{1}",
                               marca1, marca2);

            int ancho = cabecera[18] + // Convierto 4 bytes a Int32
                cabecera[19] * 256 + cabecera[20] * 256 * 256 +
                cabecera[21] * 256 * 256 * 256;
            Console.WriteLine("Ancho: {0}", ancho);

            int alto = cabecera[22] +
                cabecera[23] * 256 + cabecera[24] * 256 * 256 +
                cabecera[25] * 256 * 256 * 256;
            Console.WriteLine("Alto: {0}", alto);

            int compresion = cabecera[30];
            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;
            }

            int resolH = cabecera[38] +
                cabecera[39] * 256 + cabecera[40] * 256 * 256 +
                cabecera[41] * 256 * 256 * 256;
            Console.WriteLine("Resolución Horiz.: {0}", resolH);

            int resolV = cabecera[42] +
                cabecera[43] * 256 + cabecera[44] * 256 * 256 +
                cabecera[45] * 256 * 256 * 256;

```

```

        Console.WriteLine("Resolución Vertical: {0}", resolv);
    }
}
}
}
}

```

Ejercicios propuestos:

(8.14.1) 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.

(8.14.2) 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.

8.15. Escribir en un fichero binario

Para escribir en un `FileStream`, usaremos órdenes similares a las que empleábamos para leer de él:

- Un método `WriteByte`, para escribir sólo un byte, o bien...
- Un método `Write`, para escribir un bloque de información (desde cierta posición de un array -normalmente 0-, y con cierto tamaño).

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_08_15a.cs
// Ficheros binarios: escritura en FileStream
// Introducción a C#, por Nacho Cabanes

```

```

using System;
using System.IO;

public class Ejemplo_08_15a
{
    const int TAMANYO_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
{
    // Primero creamos el fichero, con algun dato
    fichero = File.Create( nombre );
    fichero.Write(datos, 0, 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 e)
{
    Console.WriteLine("Problemas: "+e.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_08_15b.cs
// Ficheros binarios: escritura en BinaryWriter
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_08_15b
{
    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 e)
        {
            Console.WriteLine("Problemas al crear: "+e.Message);
            return;
        }

        // Ahora vamos a leerlos
        Console.WriteLine("Leyendo fichero...");
    }
}
```

```

try
{
    ficheroEntrada = new BinaryReader(
        File.Open(nombre, FileMode.Open));

    unDatoByte = ficheroEntrada.ReadByte();
    Console.WriteLine("El byte leído es un {0}",
        unDatoByte);

    unDatoInt = ficheroEntrada.ReadInt32();
    Console.WriteLine("El int leído es un {0}",
        unDatoInt);

    unDatoFloat = ficheroEntrada.ReadSingle();
    Console.WriteLine("El float leído es un {0}",
        unDatoFloat);

    unDatoDouble = ficheroEntrada.ReadDouble();
    Console.WriteLine("El double leído es un {0}",
        unDatoDouble);

    unDatoString = ficheroEntrada.ReadString();
    Console.WriteLine("El string leído es \"{0}\"",
        unDatoString);
    Console.WriteLine("Volvamos a leer el int...");

    ficheroEntrada.BaseStream.Seek(1, SeekOrigin.Begin);
    unDatoInt = ficheroEntrada.ReadInt32();
    Console.WriteLine("El int leído es un {0}",
        unDatoInt);
    ficheroEntrada.Close();
}
catch (Exception e)
{
    Console.WriteLine("Problemas al leer: "+e.Message);
    return;
}
}
}

```

El resultado de este programa es:

```

Introduzca el nombre del fichero a crear: 1234
Creando fichero...
Leyendo fichero...
El byte leído es un 1
El int leído es un 2
El float leído es un 3
El double leído es un 4
El string leído es "Hola"
Volvamos a leer el int...
El int leído 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 se desplaza hasta el final del mismo, o crea un archivo nuevo si no existe.

Ejercicios propuestos:

(8.15.1) Crea una copia de un fichero EXE. La copia debe tener el mismo nombre y extensión BAK.

(8.15.2) Crea un programa que "encripte" el contenido de un fichero BMP, volcando su contenido a un nuevo fichero, en el que intercambiará los dos primeros bytes. Para desencriptar, bastará con volver a intercambiar esos dos bytes, volcando a un tercer fichero.

8.16. 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 encriptar un fichero sin necesidad de crear una copia, o para poder crear herramientas como 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 vio en el apartado 8.12) 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_08_16a.cs
// Lectura y escritura en fichero binario
// Introducción a C#, por Nacho Cabanes
```

```

using System;
using System.IO;

public class Ejemplo_08_16a
{
    const int TAMANYO_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.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 e)
        {
            Console.WriteLine(e.Message);
            return;
        }
    }
}

```

Ejercicios propuestos:

(8.16.1) Crea un programa que "encripte" el contenido de un fichero BMP, intercambiando los dos primeros bytes (y modificando el mismo fichero). Para desencriptar, bastará con volver a intercambiar esos dos bytes.

Ejercicios de repaso sobre ficheros:

(8.16.2) Crea 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.

(8.16.3) Un programa que vuelque todo el contenido de un fichero binario a otro, convirtiendo cada letra entre la A y la Z a mayúsculas. Los nombres de ambos ficheros se deben indicar como parámetros en la línea de comandos. Si no hay parámetros, se le preguntarán al usuario.

(8.16.4) Mejora la base de datos de ficheros (ejemplo 04_06a) para que los datos almacenados se guarden automáticamente en fichero al terminar una ejecución, y se vuelvan a cargar al principio de la siguiente.

(8.16.5) Crea un "traductor básico de C# a C", que volcará todo el contenido de un fichero de texto a otro, pero reemplazando "Console.WriteLine" con "printf", "Main" con "main", "string" con "char[80]", "Console.ReadLine" con "scanf", y eliminando "static" y "public" y las líneas que comiencen con "Using".

(8.16.6) Prepara 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.

(8.16.7) Haz un programa que duplique un fichero, copiando todo su contenido a un nuevo fichero, byte a byte. El nombre del fichero de salida se tomará del nombre del fichero de entrada, al que se añadirá ".out".

(8.16.8) Crea un programa que duplique un fichero, copiando todo su contenido a un nuevo fichero, volcando para ello todo el contenido en un array. El nombre de ambos ficheros se debe leer de la línea de comandos, o pedir al usuario en caso de no existir parámetros.

(8.16.9) Crea un programa que compare si dos ficheros son iguales byte a byte, comprobando primero su tamaño y leyendo después cada vez un byte de cada uno de ellos.

(8.16.10) Crea un programa que compare si dos ficheros son iguales, volcando todo su contenido en sendos arrays, que comparará.

(8.16.11) 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 y sobre la cabecera ID3 V1, que se encuentra -si está presente- en los últimos 128 bytes del fichero).

9. Persistencia de objetos

9.1. ¿Por qué la persistencia?

Una forma alternativa de conseguir que la información que manipula un programa esté disponible para una ejecución posterior es no guardarla en un "fichero convencional", sino pedir al sistema que se conserve el estado de los objetos que forman el programa.

Podríamos conseguirlo "de forma artesanal" si creamos un método que guarde cada uno de los atributos en un fichero y otro método que recupere cada uno de esos atributos desde el mismo fichero:

```
// Ejemplo_09_01a.cs
// Primer acercamiento a la persistencia
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Persist01
{
    int numero;

    public void SetNumero(int n)
    {
        numero = n;
    }

    public int GetNumero()
    {
        return numero;
    }

    public void Guardar(string nombre)
    {
        BinaryWriter ficheroSalida = new BinaryWriter(
            File.Open(nombre, FileMode.Create));
        ficheroSalida.Write(numero);
        ficheroSalida.Close();
    }

    public void Cargar(string nombre)
    {
        BinaryReader ficheroEntrada = new BinaryReader(
            File.Open(nombre, FileMode.Open));
        numero = ficheroEntrada.ReadInt32();
        ficheroEntrada.Close();
    }

    public static void Main()
```

```

{
    Persist01 ejemplo = new Persist01();
    ejemplo.SetNumero(5);
    Console.WriteLine("Valor: {0}", ejemplo.GetNumero());
    ejemplo.Guardar( "ejemplo.dat" );

    Persist01 ejemplo2 = new Persist01();
    Console.WriteLine("Valor 2: {0}", ejemplo2.GetNumero());

    ejemplo2.Cargar( "ejemplo.dat" );
    Console.WriteLine("Y ahora: {0}", ejemplo2.GetNumero());
}
}

```

Que daría como resultado:

```

Valor: 5
Valor 2: 0
Y ahora: 5

```

Pero esta forma de trabajar se complica mucho cuando nuestro objeto tiene muchos atributos, y más aún si se trata de bloques de objetos (por ejemplo, un "array") que a su vez contienen otros objetos. Por eso, existen maneras más automatizadas y que permiten escribir menos código.

Ejercicios propuestos:

(9.1.1) Amplía la clase Persona (ejercicio 6.2.1), para que permita guardar su estado y recuperarlo posteriormente.

9.2. Creando un objeto "serializable"

Vamos a guardar todo un objeto (incluyendo los valores de sus atributos) en un fichero. En primer lugar, tendremos que indicar "[Serializable]" antes de la clase:

```

[Serializable]
public class Persist02

```

En segundo lugar, como vamos a sobrescribir todo el objeto, en vez de sólo los atributos, ahora los métodos "Cargar" y "Guardar" ya no pueden pertenecer a esa misma clase: deberán estar en una clase auxiliar, que se encargue de salvar los datos y recuperarlos. En este primer ejemplo, nos limitaremos a declararlos "static" para que sea Main el que se encargue de esas tareas:

```

public static void Guardar(string nombre, Persist02 objeto)

```

El programa completo podría ser algo como

```
// Ejemplo_09_02a.cs
// Ejemplo básico de persistencia
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
public class Persist02
{
    int numero;

    public void SetNumero(int n)
    {
        numero = n;
    }

    public int GetNumero()
    {
        return numero;
    }

    // Métodos para guardar en fichero y leer desde él

    public static void Guardar(string nombre, Persist02 objeto)
    {
        IFormatter formatter = new BinaryFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Create, FileAccess.Write, FileShare.None);
        formatter.Serialize(stream, objeto);
        stream.Close();
    }

    public static Persist02 Cargar(string nombre)
    {
        Persist02 objeto;
        IFormatter formatter = new BinaryFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Open, FileAccess.Read, FileShare.Read);
        objeto = (Persist02)formatter.Deserialize(stream);
        stream.Close();
        return objeto;
    }

    public static void Main()
    {
        Persist02 ejemplo = new Persist02();
        ejemplo.SetNumero(5);
        Console.WriteLine("Valor: {0}", ejemplo.GetNumero());
        Guardar("ejemplo.dat", ejemplo);

        Persist02 ejemplo2 = new Persist02();
        Console.WriteLine("Valor 2: {0}", ejemplo2.GetNumero());
        ejemplo2 = Cargar("ejemplo.dat");
        Console.WriteLine("Y ahora: {0}", ejemplo2.GetNumero());
    }
}
```

```
}
```

Y su resultado sería el mismo que antes:

```
Valor: 5
Valor 2: 0
Y ahora: 5
```

Ejercicios propuestos:

(9.2.1) Crea una variante del ejercicio 9.1.1, que use serialización para guardar y recuperar los datos.

9.3. Empleando clases auxiliares

Una solución un poco más elegante podría ser encapsular la clase (o clases) que vamos a guardar, incluyéndola(s) dentro de otra clase auxiliar, de modo que podamos reutilizar esa estructura:

```
[Serializable]
public class ClaseAGuardar
{
    Ejemplo e;

    public void SetDatos(Ejemplo e1)
    {
        e = e1;
    }

    public Ejemplo GetDatos()
    {
        return e;
    }
}
```

Y crear una segunda clase, que sea la encargada de guardar y recuperar los datos:

```
public class Serializador
{
    string nombre;

    public Serializador(string nombreFich)
    {
        nombre = nombreFich;
    }

    public void Guardar(ClaseAGuardar objeto)
    {
        IFormatter formatter = new BinaryFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Create, FileAccess.Write, FileShare.None);
    }
}
```

```

        formatter.Serialize(stream, objeto);
        stream.Close();
    }

    public ClaseAGuardar Cargar()
    {
        ClaseAGuardar objeto;
        IFormatter formatter = new BinaryFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Open, FileAccess.Read, FileShare.Read);
        objeto = (ClaseAGuardar)formatter.Deserialize(stream);
        stream.Close();
        return objeto;
    }
}

```

De modo que un único fuente que contuviera estas tres clases podría ser:

```

// Ejemplo_09_03a.cs
// Ejemplo de persistencia
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

// -----
// La clase "de prueba"

[Serializable]
public class Ejemplo
{
    int numero;

    public void SetNumero(int n)
    {
        numero = n;
    }

    public int GetNumero()
    {
        return numero;
    }
}

// -----
// La clase "que realmente se va a guardar"
[Serializable]
public class ClaseAGuardar
{
    Ejemplo e;

    public void SetDatos(Ejemplo e1)
    {
        e = e1;
    }
}

```

```

    }

    public Ejemplo GetDatos()
    {
        return e;
    }
}

// -----
// La clase "encargada de guardar"

public class Serializador
{
    string nombre;

    public Serializador(string nombreFich)
    {
        nombre = nombreFich;
    }

    public void Guardar(ClasAguardar objeto)
    {
        IFormatter formatter = new BinaryFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Create, FileAccess.Write, FileShare.None);
        formatter.Serialize(stream, objeto);
        stream.Close();
    }

    public ClasAguardar Cargar()
    {
        ClasAguardar objeto;
        IFormatter formatter = new BinaryFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Open, FileAccess.Read, FileShare.Read);
        objeto = (ClasAguardar)formatter.Deserialize(stream);
        stream.Close();
        return objeto;
    }
}

// -----
// Y el programa de prueba

public class Prueba
{
    public static void Main()
    {
        Ejemplo ejemplo = new Ejemplo();
        ejemplo.SetNumero(5);
        Console.WriteLine("Valor: {0}", ejemplo.GetNumero());

        ClasAguardar guardador = new ClasAguardar();
        guardador.SetDatos(ejemplo);

        Serializador s = new Serializador("ejemplo.dat");
    }
}

```

```

        s.Guardar(guardador);

        Ejemplo ejemplo2 = new Ejemplo();
        Console.WriteLine("Valor 2: {0}", ejemplo2.GetNumero());
        ejemplo2 = s.Cargar().GetDatos();
        Console.WriteLine("Y ahora: {0}", ejemplo2.GetNumero());
    }
}

```

Vamos a comprobar que se comporta correctamente en un caso más complicado, haciendo que nuestra clase contenga varios atributos, entre ellos un array de objetos de otra clase. La clase "Serializador" no cambiará, y en un caso real apenas cambiaría la clase "ClaseAGuardar" (que aquí tampoco va a modificarse):

```

// Ejemplo_09_03b.cs
// Ejemplo de persistencia
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

// -----
// Las dos clases "de prueba": una contiene a otra

[Serializable]
public class MiniEjemplo
{
    int dato;

    public void SetDato(int n)
    {
        dato = n;
    }

    public int GetDato()
    {
        return dato;
    }
}

// -----
[Serializable]
public class Ejemplo
{
    int numero;
    float numero2;
    MiniEjemplo[] mini;

    public Ejemplo()
    {
        mini = new MiniEjemplo[100];
        for (int i=0; i<100; i++)
        {

```



```

        mini[i] = new MiniEjemplo();
        mini[i].SetDato( i*2 );
    }
}

public void SetNumero(int n)
{
    numero = n;
}

public int GetNumero()
{
    return numero;
}

public void SetMini(int n, int valor)
{
    mini[n].SetDato(valor);
}

public int GetMini(int n)
{
    return mini[n].GetDato();
}
}

// -----
// La clase "que realmente se va a guardar"

[Serializable]
public class ClaseAGuardar
{
    Ejemplo e;

    public void SetDatos(Ejemplo e1)
    {
        e = e1;
    }

    public Ejemplo GetDatos()
    {
        return e;
    }
}

// -----
// La clase "encargada de guardar"

public class Serializador
{
    string nombre;

    public Serializador(string nombreFich)
    {
        nombre = nombreFich;
    }

    public void Guardar(ClaseAGuardar objeto)

```

```

    {
        IFormatter formatter = new BinaryFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Create, FileAccess.Write, FileShare.None);
        formatter.Serialize(stream, objeto);
        stream.Close();
    }

    public ClaseAGuardar Cargar()
    {
        ClaseAGuardar objeto;
        IFormatter formatter = new BinaryFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Open, FileAccess.Read, FileShare.Read);
        objeto = (ClaseAGuardar)formatter.Deserialize(stream);
        stream.Close();
        return objeto;
    }
}

// -----
// Y el programa de prueba

public class Prueba
{
    public static void Main()
    {
        Ejemplo ejemplo = new Ejemplo();
        ejemplo.SetNumero(5);
        ejemplo.SetMini(50, 500);
        Console.WriteLine("Valor: {0}", ejemplo.GetNumero());

        ClaseAGuardar guardador = new ClaseAGuardar();
        guardador.SetDatos(ejemplo);

        Serializador s = new Serializador("ejemplo.dat");
        s.Guardar(guardador);

        Ejemplo ejemplo2 = new Ejemplo();
        Console.WriteLine("Valor 2: {0}", ejemplo2.GetNumero());
        ejemplo2 = s.Cargar().GetDatos();
        Console.WriteLine("Y ahora: {0}", ejemplo2.GetNumero());
        Console.WriteLine("dato 50: {0}", ejemplo2.GetMini(50));
    }
}

```

Esto daría como resultado:

```

Valor: 5
Valor 2: 0
Y ahora: 5
dato 50: 500

```

Ejercicios propuestos:

(9.3.1) Crea una variante del ejercicio 9.2.1, que use una clase auxiliar para la serialización.

9.4. Volcando a un fichero de texto

La forma de trabajar que estamos empleando guarda los datos en un fichero binario. Si queremos guardarlos en un fichero XML (por ejemplo para maximizar la portabilidad, garantizando que se va a leer correctamente desde algún otro sistema en el que quizá el orden de los bytes sea distinto), los cambios son mínimos: cambiar "BinaryFormatter" por "SoapFormatter", así como el correspondiente "using" y quizá añadir una DLL adicional al proyecto:

```
// Ejemplo_09_04a.cs
// Ejemplo de persistencia (XML)
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;
using System.Runtime.Serialization;
// Para la siguiente línea, puede ser necesario añadir a las referencias
// del proyecto "System.Runtime.Serialization.Formatters.Soap.dll"
using System.Runtime.Serialization.Formatters.Soap;

// -----
// Las clases "de prueba"

[Serializable]
public class MiniEjemplo
{
    int dato;

    public void SetDato(int n)
    {
        dato = n;
    }

    public int GetDato()
    {
        return dato;
    }
}

[Serializable]
public class Ejemplo
{
    int numero;
    float numero2;
    MiniEjemplo[] mini;

    public Ejemplo()
```

```

{
    mini = new MiniEjemplo[100];
    for (int i=0; i<100; i++)
    {
        mini[i] = new MiniEjemplo();
        mini[i].SetDato( i*2 );
    }
}

public void SetNumero(int n)
{
    numero = n;
}

public int GetNumero()
{
    return numero;
}

public void SetMini(int n, int valor)
{
    mini[n].SetDato(valor);
}

public int GetMini(int n)
{
    return mini[n].GetDato();
}
}

// -----
// La clase "que realmente se va a guardar"

[Serializable]
public class ClaseAGuardar
{
    Ejemplo e;

    public void SetDatos(Ejemplo e1)
    {
        e = e1;
    }

    public Ejemplo GetDatos()
    {
        return e;
    }
}

// -----
// La clase "encargada de guardar"

public class Serializador
{
    string nombre;

    public Serializador(string nombreFich)
    {

```

```

        nombre = nombreFich;
    }

    public void Guardar(ClasAguardar objeto)
    {
        IFormatter formatter = new SoapFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Create, FileAccess.Write, FileShare.None);
        formatter.Serialize(stream, objeto);
        stream.Close();
    }

    public ClasAguardar Cargar()
    {
        ClasAguardar objeto;
        IFormatter formatter = new SoapFormatter();
        Stream stream = new FileStream(nombre,
            FileMode.Open, FileAccess.Read, FileShare.Read);
        objeto = (ClasAguardar)formatter.Deserialize(stream);
        stream.Close();
        return objeto;
    }
}

// -----
// Y el programa de prueba

public class Prueba
{
    public static void Main()
    {
        Ejemplo ejemplo = new Ejemplo();
        ejemplo.SetNumero(5);
        ejemplo.SetMini(50, 500);
        Console.WriteLine("Valor: {0}", ejemplo.GetNumero());

        ClasAguardar guardador = new ClasAguardar();
        guardador.SetDatos(ejemplo);

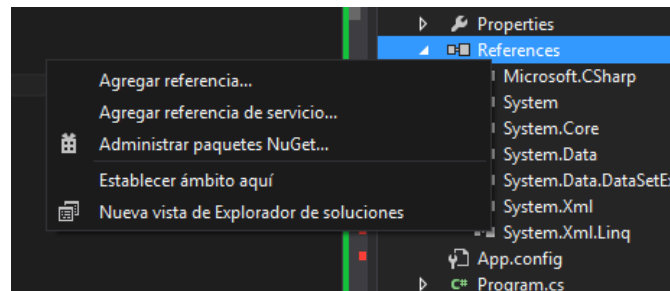
        Serializador s = new Serializador("ejemplo2.dat");
        s.Guardar(guardador);

        Ejemplo ejemplo2 = new Ejemplo();
        Console.WriteLine("Valor 2: {0}", ejemplo2.GetNumero());
        ejemplo2 = s.Cargar().GetDatos();
        Console.WriteLine("Y ahora: {0}", ejemplo2.GetNumero());
        Console.WriteLine("dato 50: {0}", ejemplo2.GetMini(50));
    }
}

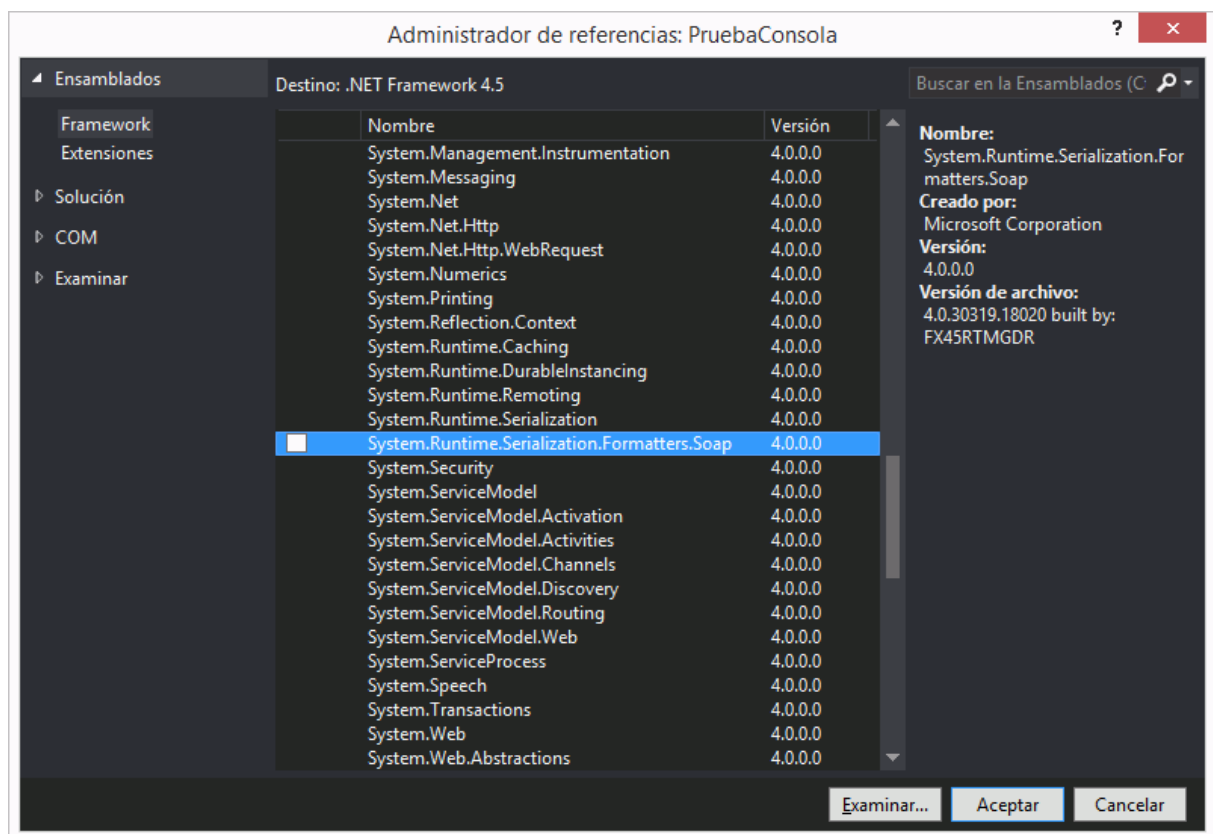
```

Este programa, que compilará correctamente en versiones de .Net como la 2.0, puede no compilar tal cual en versiones más recientes, como la 4. Si es el caso, deberemos añadir una "referencia" adicional al proyecto.

Si usamos Visual Studio, deberemos ir al panel derecho (Explorador de soluciones), hacer clic con el botón derecho en "References" e indicar que deseamos "Agregar referencia":



Dentro de esta opción, deberemos buscar dentro de las referencias que ya están previstas para nuestra versión de .Net (por ejemplo, la 4.5) el nombre "System.Runtime.Serialization.FormatterServices" y después aceptar los cambios:



Si usamos XML, el fichero de datos resultante será de mayor tamaño, pero a cambio se podrá analizar con mayor cantidad de herramientas, al ser texto puro. Por ejemplo, este es un fragmento del fichero de datos generado por el programa anterior:

```

<SOAP-ENV:Envelope      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"      xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"      xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding clr/1.0"      SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:ClaseAGuardar      id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/assem/persist05%2C%20Version%3D0.0
.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<e href="#ref-3"/>
</a1:ClaseAGuardar>
<a1:Ejemplo      id="ref-3"
xmlns:a1="http://schemas.microsoft.com/clr/assem/persist05%2C%20Version%3D0.0
.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<numero>5</numero>
<numero2>0</numero2>
<mini href="#ref-4"/>
</a1:Ejemplo>
<SOAP-ENC:Array      id="ref-4"      SOAP-ENC:arrayType="a1:MiniEjemplo[100]"
xmlns:a1="http://schemas.microsoft.com/clr/assem/persist05%2C%20Version%3D0.0
.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<item href="#ref-5"/>
<item href="#ref-6"/>
<item href="#ref-7"/>
...

<item href="#ref-102"/>
<item href="#ref-103"/>
<item href="#ref-104"/>
</SOAP-ENC:Array>
<a1:MiniEjemplo      id="ref-5"
xmlns:a1="http://schemas.microsoft.com/clr/assem/persist05%2C%20Version%3D0.0
.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<dato>0</dato>
</a1:MiniEjemplo>
<a1:MiniEjemplo      id="ref-6"
xmlns:a1="http://schemas.microsoft.com/clr/assem/persist05%2C%20Version%3D0.0
.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<dato>2</dato>
</a1:MiniEjemplo>
...

<a1:MiniEjemplo      id="ref-104"
xmlns:a1="http://schemas.microsoft.com/clr/assem/persist05%2C%20Version%3D0.0
.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<dato>198</dato>
</a1:MiniEjemplo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

(Se ha omitido la mayor parte de los datos repetitivos)

Como curiosidad, SOAP (que da su nombre al SoapFormatter) es un protocolo diseñado para poder intercambiar de forma sencilla información estructurada (como objetos y sus componentes) en aplicaciones Web.

Ejercicios propuestos:

(9.4.1) Crea una variante del ejercicio 9.3.1, que guarde los datos en formato XML.

(9.4.2) Crea una versión ampliada de la "base de datos de ficheros" (ejemplo 04_06a) para que use objetos en vez de struct, y que guarde los datos (en formato binario) usando persistencia.

(9.4.3) Añade al ejercicio de los trabajadores (6.8.1) la posibilidad de guardar sus datos en formato XML.

10. Acceso a bases de datos relacionales

10.1. Nociones mínimas de bases de datos relacionales

Una base de datos relacional es una estructura mucho más compleja (pero también más versátil) que un simple fichero de texto o binario. Hay ciertos conceptos que debemos conocer antes de empezar a trabajar con ellas:

- En una base de datos relacional hay varios bloques de datos, llamados "**tablas**" (por ejemplo, la tabla de "clientes", la tabla de "proveedores", la tabla de "productos"...).
- Cada tabla está formada por una serie de "**registros**" (por ejemplo, el cliente "Juan López", el cliente "Pedro Álvarez"...).
- Cada registro está formado por varios "**campos**" (de cada cliente podemos almacenar su nombre, su dirección postal, su teléfono, su correo electrónico, etc).
- Además, existen **relaciones** entre las tablas: por ejemplo una factura es para un cierto cliente, de modo que la tabla "factura" se relaciona con la tabla "cliente", y el nombre de esta relación es "ser para".

Vamos a aplicar estos conceptos básicos a la vez que vamos conociendo el lenguaje de consulta más habitual.

10.2. Nociones mínimas de lenguaje SQL

El lenguaje SQL (Structured Query Language) es un lenguaje muy extendido, que permite hacer consultas a una base de datos relacional. Por eso, vamos a ver algunas de las órdenes que podemos usar para almacenar y obtener datos, y posteriormente las aplicaremos desde un programa en C# que conecte a una base de datos relacional. Cuando ya seamos capaces de guardar datos y recuperarlos, veremos alguna orden más avanzada para extraer la información o para modificarla.

10.2.1. Creando la estructura

Como primer ejemplo, crearemos una base de datos sencilla, que llamaremos "ejemplo1". Esta base de datos contendrá una única tabla, llamada "agenda", que contendrá algunos datos de cada uno de nuestros amigos. Como es nuestra

primera base de datos, no pretendemos que sea perfecta, sino sencilla, así que apenas guardaremos tres datos de cada amigo: el nombre, la dirección y la edad.

Para crear la base de datos que contiene todo usaremos "create database", seguido del nombre que tendrá la base de datos:

```
create database ejemplo1;
```

En nuestro caso, nuestra base de datos almacenará una única tabla, que contendrá los datos de nuestros amigos. Por tanto, el siguiente paso será decidir qué datos concretos ("campos") guardaremos de cada amigo. Debemos pensar también qué tamaño necesitaremos para cada uno de esos datos, porque al gestor de bases de datos habrá que dárselo bastante cuadrulado. Por ejemplo, podríamos decidir lo siguiente:

```
nombre - texto, hasta 20 letras
dirección - texto, hasta 40 letras
edad - números, de hasta 3 cifras
```

Cada gestor de bases de datos tendrá una forma de llamar a esos tipos de datos. Por ejemplo, es habitual tener un tipo de datos llamado "VARCHAR" para referirnos a texto hasta una cierta longitud, y varios tipos de datos numéricos de distinto tamaño. Si usamos "INT" para indicar que nos basta con un entero no muy grande, la orden necesaria para crear esta tabla sería:

```
create table personas (
  nombre varchar(20),
  direccion varchar(40),
  edad int
);
```

10.2.2. Introduciendo datos

Para introducir datos usaremos la orden "insert", e indicaremos tras la palabra "values" los valores para los campos de texto entre comillas, y los valores para campos numéricos sin comillas, así:

```
insert into personas values ('juan', 'su casa', 25);
```

Este formato nos obliga a indicar valores para todos los campos, y exactamente en el orden en que se diseñaron. Si no queremos introducir todos los datos, o queremos hacerlo en otro orden, o no recordamos con seguridad el orden, hay

otra opción: detallar también en la orden "insert" los nombres de cada uno de los campos, así:

```
insert into personas
(nombre, direccion, edad)
values (
'pedro', 'su calle', 23
);
```

10.2.3. Mostrando datos

Para ver los datos almacenados en una tabla usaremos el formato "SELECT campos FROM tabla". Si queremos ver todos los campos, lo indicaremos usando un asterisco:

```
select * from personas;
```

que, con nuestros datos, daría como resultado (en el intérprete de comandos de algunos gestores de bases de datos, como MySQL):

```
+-----+-----+-----+
| nombre | direccion | edad |
+-----+-----+-----+
| juan   | su casa   | 25   |
| pedro  | su calle  | 23   |
+-----+-----+-----+
```

Si queremos ver sólo ciertos campos, detallamos sus nombres, separados por comas:

```
select nombre, direccion from personas;
```

y obtendríamos

```
+-----+-----+
| nombre | direccion |
+-----+-----+
| juan   | su casa   |
| pedro  | su calle  |
+-----+-----+
```

Normalmente no querremos ver todos los datos que hemos introducido, sino sólo aquellos que cumplan cierta condición. Esta condición se indica añadiendo un apartado WHERE a la orden "select", así:

```
select nombre, direccion from personas where nombre = 'juan';
```

que nos diría el nombre y la dirección de nuestros amigos llamados "juan":

```
+-----+-----+
| nombre | direccion |
+-----+-----+
|  juan  | su casa   |
+-----+-----+
```

A veces no queremos comparar con un texto exacto, sino sólo con parte del contenido del campo (por ejemplo, porque sólo sepamos un apellido o parte del nombre de la calle). En ese caso, no compararíamos con el símbolo "igual" (=), sino que usaríamos la palabra "like", y para las partes que no conozcamos usaremos el comodín "%", como en este ejemplo:

```
select nombre, direccion from personas where direccion like '%calle%';
```

que nos diría el nombre y la dirección de nuestros amigos llamados que viven en calles que contengan la palabra "calle", precedida por cualquier texto (%) y con cualquier texto (%) a continuación:

```
+-----+-----+
| nombre | direccion |
+-----+-----+
|  pedro  | su calle   |
+-----+-----+
```

10.3. Acceso a bases de datos con SQLite

SQLite es un gestor de bases de datos de pequeño tamaño, que emplea el lenguaje SQL para las consultas, y del que existe una versión que se distribuye como un fichero DLL que distribuiríamos junto al ejecutable de nuestro programa, o bien como un fichero en C que se podría integrar con los fuentes de nuestro programa... en caso de que usemos lenguaje C en nuestro proyecto.

En nuestro caso, para acceder a SQLite desde C#, tenemos disponible alguna adaptación de la biblioteca original. Una de ellas es System.Data.SQLite, que se puede descargar de <http://system.data.sqlite.org/> y tienes un pequeño proyecto de ejemplo listo para usar desde Visual Studio en <http://nachocabanes.com/csharp>

Con esta biblioteca, los pasos a seguir para crear una base de datos y guardar información en una base de datos de SQLite 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 (`new=True`) o ya existe.
- Empleando un objeto de la clase `SQLiteCommand`, detallaremos cuál es la orden SQL a ejecutar, y la lanzaremos con `ExecuteNonQuery`.
- Nos devolverá la cantidad de filas afectadas, que en nuestro caso, para una introducción de un dato, debería ser 1; si nos devuelve un dato menor que uno, nos indica que no se ha podido guardar correctamente.
- Finalmente cerraremos la conexión con `Close`:

Un fuente que dé estos pasos podría ser:

```
// EjemploSQLite1.cs
// Ejemplo de acceso a bases de datos con SQLite (1)
// Introducción a C#, por Nacho Cabanes

using System;
// Es necesario añadir la siguiente DLL a la "referencias" del proyecto
using System.Data.SQLite;

public class EjemploSQLite1
{
    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=ejemplo01.sqlite;Version=3;New=True;Compress=True;");
        conexion.Open();

        // Creamos la tabla
        string creacion = "create table personas ("
            + " nombre varchar(20),direccion varchar(40),edad int);";
        SQLiteCommand cmd = new SQLiteCommand(creacion, conexion);
        cmd.ExecuteNonQuery();

        // E insertamos dos datos
        string insercion = "insert into personas values ('juan', 'su casa',
25);";
        cmd = new SQLiteCommand(insercion, conexion);
        int cantidad = cmd.ExecuteNonQuery();
        if (cantidad < 1)
            Console.WriteLine("No se ha podido insertar");
    }
}
```

```

    insercion = "insert into personas (nombre, direccion, edad)"
                +" values ('pedro', 'su calle', 23);";
    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.");
}
}

```

Deberemos compilar con la versión 2 (o superior) de la "plataforma punto Net" y añadir el fichero "System.Data.SQLite.dll" a las "referencias" del proyecto.

Por ejemplo, con **Mono** haríamos

```
gmcs ejemploSQLite.cs /r:System.Data.SQLite.dll
```

Y para lanzar el ejecutable necesitaremos tener también la versión 2 (o superior) de la plataforma .Net, o bien lanzarlo a través de Mono:

```
mono ejemploSQLite.exe
```

Tanto el fichero "System.Data.SQLite.dll" como el "sqlite3.dll" deberán estar en la misma carpeta que nuestro ejecutable, para que éste funcione correctamente.

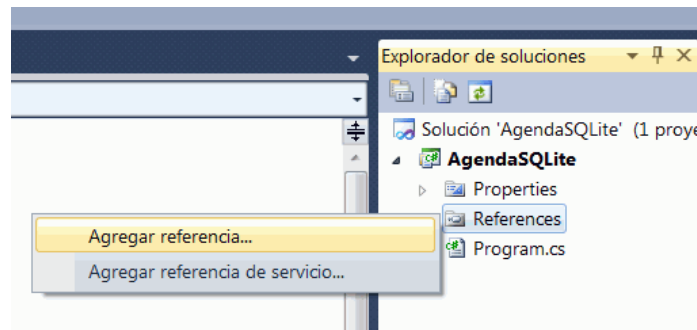
Como alternativa, también podemos crear el ejecutable usando **el compilador que incorpora la "plataforma punto Net"**, que es un fichero llamado "csc.exe" en la carpeta de la versión de la plataforma que tengamos instalada (es frecuente tener varias). Por ejemplo, para la versión 2 haríamos

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc ejemploSQLite.cs
/r:System.Data.SQLite.dll
```

Este fichero DLL es de 32 bits, de modo que si usamos una versión de Windows 64 bits puede no funcionar correctamente. La solución, si compilamos desde línea de comandos, es indicar que es para "plataforma x86", añadiendo la opción "/platform:x86", así:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc ejemploSQLite.cs
/r:System.Data.SQLite.dll /platform:x86
```

Si usamos **Visual Studio** o SharpDevelop, tendríamos que añadir ese fichero DLL a las "referencias" de nuestro proyecto. Lo podemos hacer desde la ventana del "Explorador de soluciones", pulsando el botón derecho sobre "References", "Agregar referencia" y escogiendo el fichero System.Data.SQLite.DLL desde la pestaña "Examinar":



Al igual que si compilamos desde la línea de comandos, tanto el fichero "System.Data.SQLite.dll" como el "sqlite3.dll" deberán estar en la misma carpeta que nuestro ejecutable (típicamente "bin\debug").

Como curiosidad, esta 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.

Para **mostrar los datos**, el primer y último paso serían casi iguales, pero no los intermedios:

- Crear una conexión a la base de datos, indicando en este caso que el fichero ya existe (new=false).
- Con un objeto de la clase SQLiteCommand detallaremos cuál 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:

```
// EjemploSQLite2.cs
// Ejemplo de acceso a bases de datos con SQLite (2)
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
// Es necesario añadir la siguiente DLL a la "referencias" del proyecto
using System.Data.SQLite;

public class EjemploSQLite2
{
    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=ejemplo01.sqlite;Version=3;New=False;Compress=True;");
        conexion.Open();

        // Lanzamos la consulta y preparamos la estructura para leer datos
        string consulta = "select * from personas";
        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[2]);
            // Y los mostramos
            System.Console.WriteLine("Nombre: {0}, edad: {1}",
                nombre, edad);
        }

        // Finalmente, cerramos la conexión
        conexion.Close();
    }
}
```

Que mostraría:

```
Nombre: juan, edad: 25
Nombre: pedro, edad: 23
```

Nota: en este segundo ejemplo, mostrábamos dato[0] (el primer dato, el nombre) y dato [2] (el tercer dato, la edad) pero no el dato intermedio, dato[1] (la dirección).

Ejercicios propuestos:

(10.3.1) Crea una versión de la base de datos de ficheros (ejemplo 04_06a) que realmente guarde en una base de datos (de SQLite) los datos que maneja. Deberá guardar todos antes de terminar cada sesión de uso, y volverlos a cargar al comienzo de la siguiente.

10.4. Un poco más de SQL: varias tablas

10.4.1. La necesidad de varias tablas

Puede haber varios motivos por los que nos interese trabajar con más de una tabla.

Por una parte, podemos tener bloques de información claramente distintos. Por ejemplo, en una base de datos que guarde la información de una empresa tendremos datos como los artículos que distribuimos y los clientes que nos los compran, y estos dos bloques de información no deberían guardarse en una misma tabla.

Por otra parte, habrá ocasiones en que veamos que los datos, a pesar de que se podrían clasificar dentro de un mismo "bloque de información" (tabla), serían redundantes: existiría gran cantidad de datos repetitivos, y esto puede dar lugar a dos problemas:

- Espacio desperdiciado.
- Posibilidad de errores al introducir los datos, lo que daría lugar a inconsistencias:

Veamos un ejemplo:

```
+-----+-----+-----+
| nombre | direccion | ciudad  |
+-----+-----+-----+
| juan   | su casa   | alicante |
| alberto | calle uno | alicante |
| pedro  | su calle  | alicantw |
+-----+-----+-----+
```

Si en vez de repetir "alicante" en cada una de esas fichas (registros), utilizásemos un código de ciudad, por ejemplo "a", gastaríamos menos espacio (en este ejemplo, 7 bytes menos en cada ficha).

Por otra parte, hemos tecleado mal uno de los datos: en la tercera ficha no hemos indicado "alicante", sino "allicantw", de modo que si hacemos consultas sobre personas de Alicante, la última de ellas no aparecería. Al teclear menos, es también más difícil cometer este tipo de errores.

A cambio, necesitaremos una segunda tabla, en la que guardemos los códigos de las ciudades, y el nombre al que corresponden (por ejemplo: si códigoDeCiudad = "a", la ciudad es "alicante").

10.4.2. Las claves primarias

Generalmente, será necesario tener algún dato que nos permita distinguir de forma clara los datos que tenemos almacenados. Por ejemplo, el nombre de una persona no es único: pueden aparecer en nuestra base de datos varios usuarios llamados "Juan López". Si son nuestros clientes, debemos saber cuál es cuál, para no cobrar a uno de ellos un dinero que corresponde a otro. Eso se suele solucionar guardando algún dato adicional que sí sea único para cada cliente, como puede ser el Documento Nacional de Identidad, o el Pasaporte. Si no hay ningún dato claro que nos sirva, en ocasiones añadiremos un "código de cliente", inventado por nosotros, o algo similar.

Llamaremos "claves primarias" a estos datos que distinguen claramente unas "fichas" (registros) de otras.

10.4.3. Enlazar varias tablas usando SQL

Vamos a crear la tabla de ciudades, que guardará el nombre de cada una de ellas y su código. Este código será el que actúe como "clave primaria", para distinguir otra ciudad. Por ejemplo, hay una ciudad llamado "Toledo" en España, pero también otra en Argentina, otra en Uruguay, dos en Colombia, una en Ohio (Estados Unidos)... el nombre claramente no es único, así que podríamos usar códigos como "te" para Toledo de España, "ta" para Toledo de Argentina y así sucesivamente.

La forma de crear la tabla con esos dos campos y con esa clave primaria sería:

```
create table ciudades (
  codigo varchar(3),
  nombre varchar(30),
  primary key (codigo)
);
```

Mientras que la tabla de personas sería casi igual al ejemplo anterior, pero añadiendo un nuevo dato: el código de la ciudad

```
create table personas (
  nombre varchar(20),
```

```

direccion varchar(40),
edad decimal(3),
codciudad varchar(3)
);

```

Para introducir datos, el hecho de que exista una clave primaria no supone ningún cambio, salvo por el hecho de que no se nos permitiría introducir dos ciudades con el mismo código:

```

insert into ciudades values ('a', 'alicante');
insert into ciudades values ('b', 'barcelona');
insert into ciudades values ('m', 'madrid');

insert into personas values ('juan', 'su casa', 25, 'a');
insert into personas values ('pedro', 'su calle', 23, 'm');
insert into personas values ('alberto', 'calle uno', 22, 'b');

```

Cuando queremos mostrar datos de varias tablas a la vez, deberemos hacer unos pequeños cambios en las órdenes "select" que hemos visto:

- En primer lugar, indicaremos varios nombres después de "FROM" (los de cada una de las tablas que necesitemos).
- Además, puede ocurrir que tengamos campos con el mismo nombre en distintas tablas (por ejemplo, el nombre de una persona y el nombre de una ciudad), y en ese caso deberemos escribir el nombre de la tabla antes del nombre del campo.

Por eso, una consulta básica sería algo parecido (sólo parecido) a:

```
select personas.nombre, direccion, ciudades.nombre from personas, ciudades;
```

Pero esto todavía tiene problemas: estamos combinando TODOS los datos de la tabla de personas con TODOS los datos de la tabla de ciudades, de modo que obtenemos $3 \times 3 = 9$ resultados:

nombre	direccion	nombre
juan	su casa	alicante
pedro	su calle	alicante
alberto	calle uno	alicante
juan	su casa	barcelona
pedro	su calle	barcelona
alberto	calle uno	barcelona
juan	su casa	madrid
pedro	su calle	madrid

```
| alberto | calle uno | madrid |
+-----+-----+-----+
```

Pero esos datos no son reales: si "juan" vive en la ciudad de código "a", sólo debería mostrarse junto al nombre "alicante". Nos falta indicar esa condición: "el código de ciudad que aparece en la persona debe ser el mismo que el código que aparece en la ciudad", así:

```
select personas.nombre, direccion, ciudades.nombre from personas, ciudades
where personas.codciudad = ciudades.codigo;
```

Esta será la forma en que trabajaremos normalmente. El resultado de esta consulta sería:

```
+-----+-----+-----+
| nombre | direccion | nombre |
+-----+-----+-----+
| juan   | su casa   | alicante |
| alberto | calle uno | barcelona |
| pedro  | su calle  | madrid   |
+-----+-----+-----+
```

Ese sí es el resultado correcto. Cualquier otra consulta que implique las dos tablas deberá terminar comprobando que los dos códigos coinciden. Por ejemplo, para ver qué personas viven en la ciudad llamada "madrid", haríamos:

```
select personas.nombre, direccion, edad from personas, ciudades where
ciudades.nombre='madrid' and personas.codciudad = ciudades.codigo;
```

```
+-----+-----+-----+
| nombre | direccion | edad |
+-----+-----+-----+
| pedro  | su calle  | 23 |
+-----+-----+-----+
```

Y para saber las personas de ciudades que comiencen con la letra "b", usaríamos "like":

```
select personas.nombre, direccion, ciudades.nombre from personas, ciudades
where ciudades.nombre like 'b%' and personas.codciudad = ciudades.codigo;
```

```
+-----+-----+-----+
| nombre | direccion | nombre |
+-----+-----+-----+
| alberto | calle uno | barcelona |
+-----+-----+-----+
```

Si en nuestra tabla puede haber algún dato que se repita, como la dirección, podemos pedir un listado sin duplicados, usando la palabra "distinct":

```
select distinct direccion from personas;
```

10.4.4. Varias tablas con SQLite desde C#

Vamos a crear un fuente de C# que ponga a prueba los ejemplos anteriores:

```
// EjemploSQLite3.cs
// Ejemplo de acceso a bases de datos con SQLite (3)
// Introducción a C#, por Nacho Cabanes

using System;
// Es necesario añadir la siguiente DLL a la "referencias" del proyecto
using System.Data.SQLite;

public class EjemploSQLite3
{
    public static void Crear()
    {
        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=ejemplo02.sqlite;Version=3;New=True;Compress=True;");
        conexion.Open();

        // Creamos las tablas
        Console.WriteLine(" Creando la tabla de ciudades");
        string creacion = "create table ciudades ( "
            + " codigo varchar(3), nombre varchar(30),"
            + " primary key (codigo));";
        SQLiteCommand cmd = new SQLiteCommand(creacion, conexion);
        cmd.ExecuteNonQuery();

        Console.WriteLine(" Creando la tabla de personas");
        creacion = "create table personas ( "
            + " nombre varchar(20), direccion varchar(40),"
            + " edad decimal(3), codciudad varchar(3));";
        cmd = new SQLiteCommand(creacion, conexion);
        cmd.ExecuteNonQuery();

        // E insertamos datos
        Console.WriteLine(" Introduciendo ciudades");
        string insercion = "insert into ciudades values "
            + "('a', 'alicante');";
        cmd = new SQLiteCommand(insercion, conexion);
        int cantidad = cmd.ExecuteNonQuery();
        if (cantidad < 1)
            Console.WriteLine("No se ha podido insertar");
    }
}
```

```

insercion = "insert into ciudades values "
            + "('b', 'barcelona');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

insercion = "insert into ciudades values "
            + "('m', 'madrid');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

Console.WriteLine(" Introduciendo personas");
insercion = "insert into personas values "
            + "('juan', 'su casa', 25, 'a');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

insercion = "insert into personas values "
            + "('pedro', 'su calle', 23, 'm');";
cmd = new SQLiteCommand(insercion, conexion);
cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");

insercion = "insert into personas values "
            + "('alberto', 'calle uno', 22, 'b');";
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("Base de datos creada.");
}

public static void Mostrar()
{
    // Creamos la conexion a la BD
    // El Data Source contiene la ruta del archivo de la BD
    SQLiteConnection conexion =
        new SQLiteConnection
            ("Data
Source=ejemplo02.sqlite;Version=3;New=False;Compress=True;");
    conexion.Open();

    // Lanzamos la consulta y preparamos la estructura para leer datos
    string consulta = "select personas.nombre, direccion, ciudades.nombre
    "
        + "from personas, ciudades where personas.codciudad =
ciudades.codigo;";
    SQLiteCommand cmd = new SQLiteCommand(consulta, conexion);
    SQLiteDataReader datos = cmd.ExecuteReader();

```

```

Console.WriteLine("Datos:");
// Leemos los datos de forma repetitiva
while (datos.Read())
{
    // Y los mostramos
    Console.WriteLine("  {0} - {1} - {2}",
        Convert.ToString(datos[0]), Convert.ToString(datos[1]),
        Convert.ToString(datos[2]));
}

// Finalmente, cerramos la conexión
conexion.Close();
}

public static void Main()
{
    Crear();
    Mostrar();
}
}

```

Su resultado sería:

```

Creando la base de datos...
  Creando la tabla de ciudades
  Creando la tabla de personas
  Introduciendo ciudades
  Introduciendo personas
Base de datos creada.
Datos:
  juan - su casa - alicante
  pedro - su calle - madrid
  alberto - calle uno - barcelona

```

Ejercicios propuestos:

(10.4.1) Mejora el ejercicio 10.3.1 para que, además de el nombre del fichero y su tamaño, guarde una categoría (por ejemplo, "utilidad" o "vídeo"). Estas categorías estarán almacenadas en una segunda tabla.

10.5. Borrado y modificación de datos

Podemos **borrar** los datos que cumplen una cierta condición. La orden es "delete from", y con "where" indicamos las condiciones que se deben cumplir, de forma similar a como hacíamos en la orden "select":

```
delete from personas where nombre = 'juan';
```

Esto borraría todas las personas llamadas "juan" que estén almacenadas en la tabla "personas".

Cuidado: si no se indica la parte de "where", no se borrarían los datos que cumplen una condición, sino TODOS los datos.

Para **modificar** datos de una tabla, el formato habitual es "update tabla set campo=nuevoValor where condicion".

Por ejemplo, si hemos escrito "Alberto" en minúsculas ("alberto"), lo podríamos corregir con:

```
update personas set nombre = 'Alberto' where nombre = 'alberto';
```

Y si queremos corregir todas las edades para sumarles un año se haría con

```
update personas set edad = edad+1;
```

(al igual que habíamos visto para "select" y para "delete", si no indicamos la parte del "where", los cambios se aplicarán a todos los registros de la tabla).

Ejercicios propuestos:

(10.5.1) Crea una versión del ejercicio 10.5.1 que no guarde todos los datos al salir, sino que actualice con cada nueva modificación: inserte los nuevos datos inmediatamente, permita borrar un registro (reflejando los cambios inmediatamente) y modificar los datos de un registro (ídem).

10.6. Operaciones matemáticas con los datos

Desde SQL podemos realizar operaciones a partir de los datos antes de mostrarlos. Por ejemplo, podemos mostrar cuál era la edad de una persona hace un año, con

```
select edad-1 from personas;
```

Los operadores matemáticos que podemos emplear son los habituales en cualquier lenguaje de programación, ligeramente ampliados: + (suma), - (resta y negación), * (multiplicación), / (división) . La división calcula el resultado con decimales; si queremos trabajar con números enteros, también tenemos los operadores DIV (división entera) y MOD (resto de la división):


```
select 5/2, 5 div 2, 5 mod 2;
```

Darí­a como resultado

```
+-----+-----+-----+
| 5/2    | 5 div 2 | 5 mod 2 |
+-----+-----+-----+
| 2.5000 |        2 |        1 |
+-----+-----+-----+
```

También podemos aplicar ciertas funciones matemáticas **a todo un conjunto de datos** de una tabla. Por ejemplo, podemos saber cuál es la edad más baja de entre las personas que tenemos en nuestra base de datos, haríamos:

```
select min(edad) from personas;
```

Las "funciones de agregación" más habituales son:

- min = mínimo valor
- max = máximo valor
- sum = suma de los valores
- avg = media de los valores
- count = cantidad de valores

La forma más habitual de usar "count" es pidiendo con "count(*)" que se nos muestren todos los datos que cumplen una condición. Por ejemplo, podríamos saber cuántas personas tienen una dirección que comience por la letra "s", así:

```
select count(*) from personas where direccion like 's%';
```

10.7 Grupos

Puede ocurrir que no nos interese un único valor agrupado para todos los datos (el total, la media, la cantidad de datos), sino el resultado para un grupo de datos. Por ejemplo: saber no sólo la cantidad de clientes que hay registrados en nuestra base de datos, sino también la cantidad de clientes que viven en cada ciudad.

La forma de obtener subtotales es creando grupos con la orden "group by", y entonces pidiendo una valor agrupado (count, sum, avg, ...) para cada uno de esos grupos. Por ejemplo, en nuestra tabla "personas", podríamos saber cuántas personas aparecen de cada edad, con:

```
select count(*), edad from personas group by edad;
```

que daría como resultado

count(*)	edad
1	22
1	23
1	25

Pero podemos llegar más allá: podemos no trabajar con todos los grupos posibles, sino sólo con los que cumplen alguna condición.

La condición que se aplica a los grupos no se indica con "where", sino con "having" (que se podría traducir como "los que tengan..."). Un ejemplo:

```
select count(*), edad from personas group by edad having edad > 24;
```

que mostraría

count(*)	edad
1	25

En el lenguaje SQL existe mucho más que lo que hemos visto aquí, pero para nuestro uso desde C# y SQLite debería ser suficiente.

Ejercicios propuestos:

(10.7.1) Crea una versión del ejercicio 10.4.1 que permita saber cuántos ficheros hay pertenecientes a cada categoría.

10.8 Un ejemplo completo con C# y SQLite

Vamos a crear un pequeño ejemplo que, para una única tabla, permita añadir datos, mostrar todos ellos o buscar los que cumplan una cierta condición:

```
// AgendaSQLite.cs
// Ejemplo de acceso a bases de datos con SQLite: agenda
// Introducción a C#, por Nacho Cabanes
```

```

using System;
using System.IO; // Para File.Exists

// Es necesario añadir la siguiente DLL a la "referencias" del proyecto
using System.Data.SQLite;

public class AgendaSQLite
{
    static SQLiteConnection conexion;

    // Constructor
    public AgendaSQLite()
    {
        AbrirBD();
    }

    // Abre la base de datos, o la crea si no existe
    private void AbrirBD()
    {
        if (!File.Exists("agenda.sqlite"))
        {
            // Si no existe, creamos la base de datos
            conexion = new SQLiteConnection (
                "Data
Source=agenda.sqlite;Version=3;New=True;Compress=True;");
            conexion.Open();

            // Y creamos la tabla
            string creacion = "CREATE TABLE persona "
                + "(nombre VARCHAR(30), direccion VARCHAR(40), "
                + " edad INT );";
            SQLiteCommand cmd = new SQLiteCommand(creacion, conexion);
            cmd.ExecuteNonQuery();
        }
        else
        {
            // Si ya existe, abrimos
            conexion = new SQLiteConnection(
                "Data
Source=agenda.sqlite;Version=3;New=False;Compress=True;");
            conexion.Open();
        }
    }

    public bool InsertarDatos(string nombre, string direccion, int edad)
    {
        string insercion; // Orden de insercion, en SQL
        SQLiteCommand cmd; // Comando de SQLite
        int cantidad;      // Resultado: cantidad de datos

        try
        {
            insercion = "INSERT INTO persona " +
                "VALUES ('"+nombre+"', '"+direccion+"', "+edad+");";
            cmd = new SQLiteCommand(insercion, conexion);
            cantidad = cmd.ExecuteNonQuery();
            if (cantidad < 1)
                return false; // Si no se ha podido insertar
        }
    }
}

```

```

        catch (Exception e)
        {
            return false; // Si no se ha podido insertar (codigo repetido?)
        }
        return true; // Si todo ha ido bien, devolvemos true
    }

    // Leer todos los datos y devolverlos en un string de varias líneas
    public string LeerTodosDatos()
    {
        // Lanzamos la consulta y preparamos la estructura para leer datos
        string consulta = "select * from persona";
        string resultado = "";
        SQLiteCommand cmd = new SQLiteCommand(consulta, conexion);
        SQLiteDataReader datos = cmd.ExecuteReader();
        // Leemos los datos de forma repetitiva
        while (datos.Read())
        {
            resultado += Convert.ToString(datos[0]) + " - "
                + Convert.ToString(datos[1]) + " - "
                + Convert.ToInt32(datos[2]) + "\n";
        }
        return resultado;
    }

    // Leer todos los datos y devolverlos en un string de varias líneas
    public string LeerBusqueda(string texto)
    {
        // Lanzamos la consulta y preparamos la estructura para leer datos
        string consulta = "select * from persona where nombre like '%"
            + texto + "%' or direccion like '%" + texto + "%'";
        string resultado = "";
        SQLiteCommand cmd = new SQLiteCommand(consulta, conexion);
        SQLiteDataReader datos = cmd.ExecuteReader();
        // Leemos los datos de forma repetitiva
        while (datos.Read())
        {
            resultado += Convert.ToString(datos[0]) + " - "
                + Convert.ToString(datos[1]) + " - "
                + Convert.ToInt32(datos[2]) + "\n";
        }
        return resultado;
    }

    ~AgendaSQLite()
    {
        conexion.Close();
    }
}

// -----

public class pruebaSQLite01
{
    public static void Main()
    {
        AgendaSQLite agenda = new AgendaSQLite();
        string opcion;
    }
}

```

```

do
{
    Console.WriteLine("Escoja una opción...");
    Console.WriteLine("1.- Añadir");
    Console.WriteLine("2.- Ver todos");
    Console.WriteLine("3.- Buscar");
    Console.WriteLine("0.- Salir");

    opcion = Console.ReadLine();

    switch (opcion)
    {
        case "1":
            Console.Write("Nombre? ");
            string n = Console.ReadLine();
            Console.Write("Dirección? ");
            string d = Console.ReadLine();
            Console.Write("Edad? ");
            int e = Convert.ToInt32(Console.ReadLine());
            agenda.InsertarDatos(n, d, e);
            break;
        case "2":
            Console.WriteLine(agenda.LeerTodosDatos());
            break;
        case "3":
            Console.Write("Texto a buscar? ");
            string txt = Console.ReadLine();
            Console.WriteLine(agenda.LeerBusqueda(txt));
            break;
    }
} while (opcion != "0");
}
}

```

Ejercicios propuestos:

(10.8.1) Amplía este ejemplo (AgendaSQLite), para que se pueda borrar un dato a partir de su nombre (que debe coincidir exactamente).

(10.8.2) Amplía el ejemplo 10.8.1, para que se pueda modificar un registro.

(10.8.3) Amplía el ejemplo 10.8.2, para que permita exportar los datos a un fichero de texto (que contenga el nombre, la dirección y la edad correspondientes a cada registro en líneas separados), o bien se pueda importar datos desde un fichero de texto (añadiendo al final de los existentes).

(10.8.4) Amplía el ejemplo 10.8.2 con una tabla de ciudades, que se deberá pedir (y mostrar) de forma independiente al resto de la dirección.

10.9. Nociones mínimas de acceso desde un entorno gráfico

En la mayoría de los casos, cuando trabajamos desde un entorno "de ventanas" (en un apéndice de este texto verás las nociones básicas sobre cómo crearlos), tendremos a nuestra disposición ciertos componentes visuales que nos permitan "navegar" por los datos de forma muy simple.

Por ejemplo, existe un "DataGridView", que nos permite recorrer los datos en una vista de tabla, y poder hacer modificaciones sobre ellos:



Pero esto lo veremos (con poco detalle) más adelante, cuando entremos en contacto con el entorno gráfico conocido como "Windows Forms".

11. Punteros y gestión dinámica de memoria

11.1. *¿Por qué usar estructuras dinámicas?*

Hasta ahora teníamos una serie de variables que declarábamos 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.

Por ejemplo, si queremos crear una agenda, necesitaremos ir añadiendo nuevos datos. Si reservamos espacio para un máximo de 10 fichas, no podremos llegar a añadir la número 11. 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. Otra (mala) solución sería la de trabajar siempre en el disco y usar acceso aleatorio para leer cada ficha desde disco cuando sea necesario, pero esta alternativa es muchísimo más lenta.

La solución real 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**. Una "pila de datos" se comportará de forma similar a una pila de libros: podemos apilar cosas en la cima, o extraer 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**. Una "cola de datos" se comportará 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**, mas versátiles pero más complejas de programar, en las que se puede añadir elementos en cualquier posición y obtenerlos o borrarlos de cualquier posición.

Y existen otras estructuras dinámicas más complejas y que nosotros no trataremos, como los **árboles**, en los que cada elemento puede tener varios sucesores (se parte de un elemento "raíz", y la estructura se va ramificando), o los **grafos**, formados por una serie de nodos unidos por aristas.

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 (máximo) prefijado. Además, ciertas operaciones, como las ordenaciones o los borrados, pueden ser más rápidas que en un array.

Veremos ejemplos de cómo crear estructuras dinámicas de estos tipos en C#, usando las posibilidades que incluye la plataforma ".Net" y después comentaremos los pasos para crear una estructura dinámica de forma "artesanal".

11.2. Una pila en C#

Para crear una pila, emplearemos la clase Stack. Una pila nos permitirá 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 denotar también usando 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_11_02a.cs
// Ejemplo de clase "Stack" (Pila)
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```



```

using System.Collections;

public class Ejemplo_11_02a
{
    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,

```

Como se puede ver en este ejemplo, no hemos indicado que sea una "pila de strings", sino simplemente "una pila". Por eso, los datos que extraemos son "objetos", que deberemos convertir al tipo de datos que nos interese utilizando un "typecast" (conversión forzada de tipos), como en `palabra = (string) miPila.Pop();`

La implementación de una pila en C# es algo más avanzada que lo que podríamos esperar en una pila estándar: incorpora 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.

Ejercicios propuestos:

(11.2.1) Crea un programa que pida al usuario 5 números enteros y luego los muestre en orden contrario, utilizando una pila.

(11.2.2) Crea un programa que pida al usuario el nombre de un fichero de texto y muestre en orden inverso las líneas que lo forma, empleando una pila.

(11.2.3) Crea una clase que imite el comportamiento de una pila, pero usando internamente un array (si no lo consigues, no te preocupes; en un apartado posterior veremos una forma de hacerlo).

(11.2.4) La "notación polaca inversa" es una forma de expresar operaciones que consiste en indicar los operandos antes del correspondiente operador. Por ejemplo, en vez de "3+4" se escribiría "3 4 +". Es una notación que no necesita paréntesis y que se puede resolver usando una pila: si se recibe un dato numérico, éste se guarda en la pila; si se recibe un operador, se obtienen los dos operandos que hay en la cima de la pila, se realiza la operación y se apila su resultado. El proceso termina cuando sólo hay un dato en la pila. Por ejemplo, "3 4 +" se convierte en: apilar 3, apilar 4, sacar dos datos y sumarlos, guardar 7, terminado. Implementalo y comprueba si el resultado de "3 4 6 5 - + * 6 +" es 21.

11.3. 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_11_03a.cs
// Ejemplo de clase "Queue" (Cola)
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

public class Ejemplo_11_03a
{
    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 );
    }
}

```

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.

Ejercicios propuestos:

(11.3.1) Crea un programa que pida al usuario 5 números reales de doble precisión, los guarde en una cola y luego los muestre en pantalla.

(11.3.2) Crea un programa que pida frases al usuario, hasta que introduzca una frase vacía. En ese momento, mostrará todas las frases que se habían introducido.

(11.3.3) Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en una cola, luego muestre este contenido en pantalla y finalmente lo vuelque a otro fichero de texto.

11.4. 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 a cuyos elementos se puede acceder como a los de un array ("ArrayList") y una lista ordenada ("SortedList").

11.4.1. 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 incluso ordenar toda la lista con "Sort". Vamos a ver un ejemplo de la mayoría de sus posibilidades:

```
// Ejemplo_11_04_01a.cs
// Ejemplo de ArrayList
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

public class Ejemplo_11_04_01a
{
    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 quizá 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 quisiéramos insertar la frase "Hasta Luego", su posición correcta para que todo el ArrayList permaneciera ordenado sería la 1, que viene indicada por "~posicion".

Veremos los operadores a nivel de bits, como ~, en el tema 13, que estará dedicado a otras características avanzadas de C#.

Ejercicios propuestos:

(11.4.1.1) Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en un ArrayList, y luego pregunte de forma repetitiva al usuario qué línea desea ver. Terminará cuando el usuario introduzca "-1".

(11.4.1.2) Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en un ArrayList, y luego pregunte de forma repetitiva al usuario qué texto desea buscar y muestre las líneas que contienen ese texto. Terminará cuando el usuario introduzca una cadena vacía.

(11.4.1.3) Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en un ArrayList, lo ordene y lo muestre ordenado en pantalla.

(11.4.1.4) Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en un ArrayList, luego muestre en pantalla las líneas

impares (primera, tercera, etc.) y finalmente vuelque a otro fichero de texto las líneas pares (segunda, cuarta, etc.).

(11.4.1.5) Crea una nueva versión de la "bases de datos de ficheros" (ejemplo 04_06a), pero usando ArrayList en vez de un array convencional.

11.4.2. 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 (con corchetes), como en este ejemplo:

```
// Ejemplo_11_04_02a.cs
// Ejemplo de SortedList: Diccionario esp-ing
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

public class Ejemplo_11_04_02a
{
    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") ));
        Console.WriteLine( "Que también se puede obtener con corchetes: {0}",
            miDiccio["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.

Ejercicios propuestos:

(11.4.2.1) Crea un programa que, cuando el usuario introduzca el nombre de un número del 1 al 10 en inglés (por ejemplo, "two"), diga su traducción en español (por ejemplo, "dos").

(11.4.2.2) Crea un traductor básico de C# a Pascal, que tenga las traducciones almacenadas en una SortedList (por ejemplo, "{" se convertirá a "begin", "}" se convertirá a "end", "WriteLine" se convertirá a "WriteLn", "ReadLine" se convertirá a "ReadLn", "void" se convertirá a "procedure" y "Console." se convertirá a una cadena vacía. Úsalo para convertir un abrir un fichero que contenga un fuente en C# y mostrar su equivalente (que no será perfecto) en Pascal.

11.5. 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 y ocupan más espacio, 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_11_05a.cs
// Ejemplo de Tabla Hash: Diccionario de informática
// Introducción a C#, por Nacho Cabanes

```

```

using System;
using System.Collections;

public class Ejemplo_11_05a
{

```



```

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)
    {
        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_11_05b.cs
// Ejemplo de Tabla Hash: Diccionario de informática
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

public class Ejemplo_11_05b
{
    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);
```

Ejercicios propuestos:

(11.5.1) Crea una versión alternativa del ejercicio 11.4.2.1, pero que tenga las traducciones almacenadas en una tabla Hash.

(11.5.2) Crea una versión alternativa del ejercicio 11.4.2.2, pero que tenga las traducciones almacenadas en una tabla Hash.

11.6. 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_11_06a.cs
// Ejemplo de Enumeradores en una Tabla Hash
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

public class Ejemplo_11_06a
{
    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_11_06b.cs
// Ejemplo de Enumeradores en una pila (Stack)
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections;

public class Ejemplo_11_06b
{
    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 (muchas de la cual están 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#)

11.7. *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 (inicialmente 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_11_07a.cs
// Ejemplo de clase "Pila" basada en un array
// Introducción a C#, por Nacho Cabanes

```

```
using System;
```

```

public class PilaString
{
    string[] datosPila;
    int posicionPila;
    const int MAXPILA = 200;

    // 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;
    }

    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 );
        }
    } // Fin da Main de prueba
} // Fin de la clase

```

Ejercicios propuestos:

(11.7.1) Usando esta misma estructura de programa, crea una clase "Cola", que permita introducir datos (números enteros) 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".

(11.7.2) 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"). Deberá almacenar "strings".

(11.7.3) Crea una pila de "doubles", usando internamente un ArrayList en vez de un array.

(11.7.4) Crea una cola que almacene un bloque de datos (struct, con los campos que tú elijas) usando un ArrayList.

(11.7.5) Crea una lista ordenada (de "strings") usando un ArrayList.

11.8. Introducción a los "generics"

Una ventaja, pero también a la vez un inconveniente, de las estructuras dinámicas que hemos visto, es que permiten guardar datos de cualquier tipo, incluso datos de distinto tipo en una misma estructura: un ArrayList que contenga primero un string, luego un número entero, luego uno de coma flotante, después un struct... Esto obliga a que hagamos una "conversión de tipos" con cada dato que obtenemos (excepto con los "strings").

En ocasiones puede ser interesante algo un poco más rígido, que con las ventajas de un ArrayList (crecimiento dinámico, múltiples métodos disponibles) esté adaptado a un tipo de datos, y no necesite una conversión de tipos cada vez que extraigamos un dato.

Por ello, en la versión 2 de la "plataforma .Net" se introdujeron los "generics", que definen estructuras de datos genéricas, que nosotros podemos particularizar en cada uso. Por ejemplo, una **lista** de strings se definiría con:

```
List<string> miLista = new List<string>();
```

Y necesitaríamos incluir un nuevo "using" al principio del programa:

```
using System.Collections.Generic;
```

Con sólo estos dos cambios, el ejemplo de uso de ArrayList que vimos en el apartado 11.4.1 funcionaría perfectamente:

```
// Ejemplo_11_08a.cs
// Ejemplo de List<string>
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

public class Ejemplo_11_08a
{
    public static void Main()
    {
        List<string> miLista = new List<string>();
        // 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] );
    }
}

```

De esta misma forma, podríamos crear una lista de structs, o de objetos, o de cualquier otro dato.

No sólo tenemos listas. Por ejemplo, también existe un tipo "**Dictionary**", que equivale a una tabla Hash, pero en la que las claves y los valores no tienen por qué ser strings, sino el tipo de datos que nosotros decidamos. Por ejemplo, podemos usar una cadena como clave, pero un número entero como valor obtenido:

```
Dictionary<string, int> dict = new Dictionary<string, int>();
```

Así, con un diccionario que tenga tanto claves string como valores string, podríamos crear una versión alternativa del ejemplo 11_05b. Los únicos cambios serían una declaración parecida a la anterior, el "using" correcto, y cambiar Contains por ContainsKey:

```

// Ejemplo_11_08b.cs
// Ejemplo de Dictionary
// Introducción a C#, por Nacho Cabanes

using System;
using System.Collections.Generic;

public class Ejemplo_11_08b
{
    public static void Main()
    {
        // Creamos e insertamos datos
        Dictionary<string, string> miDiccio = new Dictionary<string,
string>();
        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.ContainsKey("pc"))
    Console.WriteLine( "El significado de PC es: {0}",
        miDiccio["pc"]);
else
    Console.WriteLine( "No existe la palabra PC");
}
}

```

Ejercicios propuestos:

(11.8.1) Crea una nueva versión de la "bases de datos de ficheros" (ejemplo 04_06a), pero usando List en vez de un array convencional.

(11.8.2) Crea un programa que lea todo el contenido de un fichero, lo guarde en una lista de strings y luego lo muestre en orden inverso (de la última línea a la primera).

11.9. Los punteros en C#

11.9.1. ¿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, escribiremos un asterisco entre el tipo de datos y el nombre de la variable. Ese asterisco puede ir junto a cualquiera de ambos, por lo que 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;
```

11.9.2. 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() { ...
```

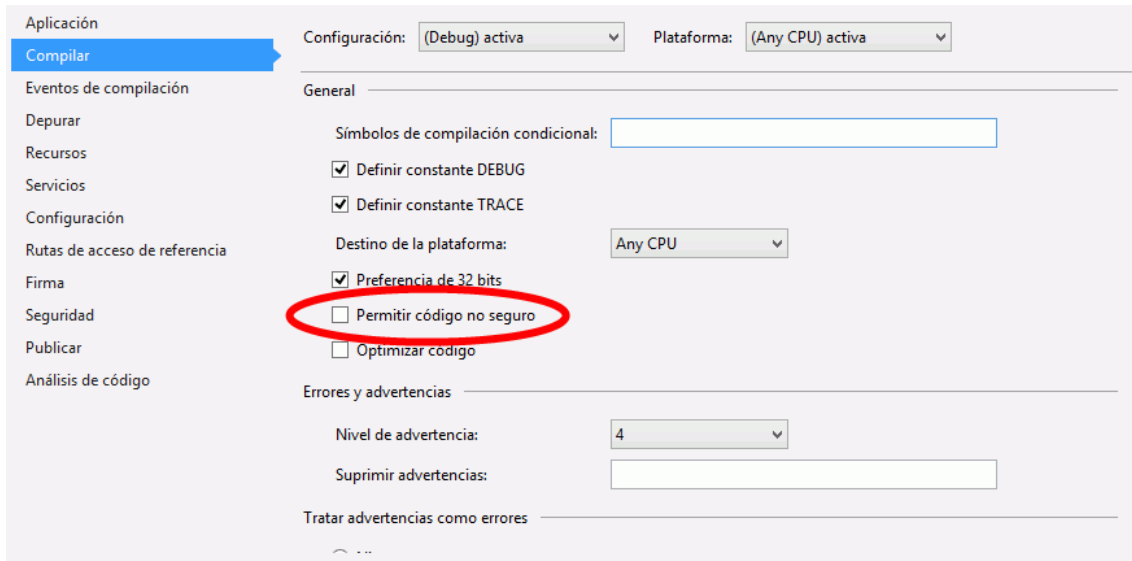
Es más, si intentamos compilar obtendremos un mensaje de error, que nos dice que si nuestro código no es seguro, deberemos compilarlo con la opción "unsafe":

```
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
```

Si usamos Visual Studio como herramienta de desarrollo, tendremos que ir a las "propiedades del proyecto" (típicamente estará al final del menú "Proyecto") y decir que queremos "Permitir código no seguro":



11.9.3. Uso básico de punteros

Veamos un ejemplo básico de cómo dar valor a un puntero y de cómo guardar un valor en él:

```
// Ejemplo_11_09_03a.cs
// Primer ejemplo de punteros
// Introducción a C#, por Nacho Cabanes
```

```
using System;
```

```
public class Ejemplo_11_09_03a
{
    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.

Ejercicios propuestos:

(11.9.3.1) Crea un programa que intercambie la posición de memoria en la que se encuentran dos variables y que luego muestre sus contenidos en pantalla, para comprobar que no son los mismos que al principio.

11.9.4. Zonas inseguras

También podemos hacer que sólo una parte de un programa sea insegura, indicando entre llaves una parte de una función:

```

// Ejemplo_11_09_04a.cs
// Segundo ejemplo de punteros
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_11_09_04a
{
    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);
}

```

11.9.5. 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_11_09_05a.cs
// Tercer ejemplo de punteros: stackalloc
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_11_09_05a
{
    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.

Ejercicios propuestos:

(11.9.5.1) Crea una programa que pida al usuario 5 strings, los almacene usando "stackalloc" y luego los muestre en orden inverso.

11.9.6. 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_11_09_06a.cs
// Cuarto ejemplo de punteros: aritmética de punteros
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_11_09_06a
{
    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

```

Ejercicios propuestos:

(11.9.6.1) Crea una programa que pida al usuario 4 números reales, los almacene usando "stackalloc" y luego los recorra incrementando el puntero a partir de la posición del primer elemento.

11.9.7. 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_11_09_07a.cs
// Quinto ejemplo de punteros: fixed
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_11_09_07a
{
    public unsafe static void Main()
    {
        int[] datos={10,20,30};

        Console.WriteLine("Leyendo el segundo dato...");
        fixed (int* posicionDato = &datos[1])
        {
            Console.WriteLine("En posicionDato hay {0}",

```

```

        *posicionDato);
    }

    Console.WriteLine("Leyendo el primer dato...");
    fixed (int* posicionDato = datos)
    {
        Console.WriteLine("Ahora en posicionDato hay {0}",
            *posicionDato);
    }
}

```

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

```


12. Algunas bibliotecas adicionales de uso frecuente

12.1. 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_12_01a.cs
// Ejemplo básico de manejo de fechas
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_12_01a
{
    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);
    }
}
```

Algunos de las propiedades más útiles de este tipo de datos 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, que se puede convertir en un número del 0-domingo- al 6-sábado- si se fuerza su tipo a entero, anteponiéndole `"(int)"`); `DayOfYear` (día del año).

Para calcular nuevas fechas, podemos usar métodos que generan un nuevo `DateTime`, como: `AddDays` (que aparece en el ejemplo anterior), `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_12_01b.cs
// Diferencia entre dos fechas
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_12_01b
{
    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 8886 días
Si lo quieres con decimales: 8886,41919806277 días
Y si quieres más detalles: 8886.10:03:38.7126236
```

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 de "using", o bien usar la llamada completa:

```
// Ejemplo_12_01c.cs
// Pausas
// Introducción a C#, por Nacho Cabanes

using System;
using System.Threading;

public class Ejemplo_12_01c
{
    public static void Main()
    {
        DateTime fecha = DateTime.Now;
        Console.WriteLine("Son las {0}:{1}:{2}",
            fecha.Hour, fecha.Minute, fecha.Second);
        Console.WriteLine("Vamos a esperar 3 segundos...");
        Thread.Sleep(3000);
        fecha = DateTime.Now;
        Console.WriteLine("Ahora son las {0}:{1}:{2}",
            fecha.Hour, fecha.Minute, fecha.Second);
    }
}
```

Ejercicios propuestos:

(12.1.1) Crea una versión mejorada del ejemplo 12_01a, que muestre el nombre del mes (usa un array para almacenar los nombres y accede a la posición correspondiente), la hora, los minutos, los segundos y las décimas de segundo (no las milésimas). Si los minutos o los segundos son inferiores a 10, deberán mostrarse con un cero inicial, de modo que siempre aparezcan con dos cifras.

(12.1.2) Crea un reloj que se muestre en pantalla, y que se actualice cada segundo (usando "Sleep"). En esta primera aproximación, el reloj se escribirá con "WriteLine", de modo que aparecerá en la primera línea de pantalla, luego en la segunda, luego en la tercera y así sucesivamente (en el próximo apartado veremos cómo hacer que se mantenga fijo en unas ciertas coordenadas de la pantalla).

(12.1.3) Crea un programa que pida al usuario su fecha de nacimiento, y diga de qué día de la semana se trataba.

(12.1.4) Crea un programa que muestre el calendario del mes actual (pista: primero deberás calcular qué día de la semana es el día 1 de este mes). Deberá ser algo como:

```
Mayo 2015

lu ma mi ju vi sa do
          1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28
```

(12.1.5) Crea un programa que muestre el calendario correspondiente al mes y el año que se indiquen como parámetros en línea de comandos.

(12.1.6) Crea un programa que vuelque a un fichero de texto el calendario del año que se indique como parámetro en línea de comandos. Deben mostrarse tres meses en anchura: el primer bloque contendrá enero febrero y marzo, el segundo tendrá abril, mayo y junio y así sucesivamente.

12.2. Más posibilidades de la "consola"

En "Console" hay mucho más que ReadLine y WriteLine, aunque no todas las posibilidades están incluidas en las primeras versiones de la "plataforma punto net". Vamos a ver algunas de las funcionalidades adicionales 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_12_02a.cs
// Más posibilidades de "System.Console"
// Introducción a C#, por Nacho Cabanes
```

```
using System;

public class Ejemplo_12_02a
{
    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
    Console.SetCursorPosition(posX, posY);
    Console.WriteLine("Bienvenido");

    Console.ForegroundColor = ConsoleColor.Blue;
    Console.SetCursorPosition(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;
    Console.SetCursorPosition(maxX-50, maxY-1);
    Console.ForegroundColor = ConsoleColor.Red;
    Console.Write("Pulsa una tecla para terminar... ");
    Console.ReadKey(true);
}
}

```

(Nota: no todas las posibilidades que se aplican en este ejemplo están disponibles en la plataforma .Net 1.x, sino a partir de la versión 2).

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 IMPRPANT), 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 Suma), 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");
```

Console.ReadKey hace que el programa se quede **parado** hasta que se pulse una tecla. Si queremos hacer algo mientras que el usuario no pulse ninguna tecla, podemos emplear Console.KeyAvailable para comprobar si ya se ha pulsado alguna tecla que haya que analizar, como en este ejemplo, que permite mover un símbolo a izquierda y derecha, pero muestra una animación simple si no pulsamos ninguna tecla:

```
// Ejemplo_12_02b.cs
// No bloquear el programa con Console.ReadKey
// Introducción a C#, por Nacho Cabanes
```

```

using System;
using System.Threading;

public class Ejemplo_12_02b
{
    public static void Main()
    {
        int posX=40, posY=10;
        string simbolos = "^>v<";
        byte simboloActual = 0;
        bool terminado = false;

        do
        {
            Console.Clear();
            Console.SetCursorPosition(posX, posY);
            Console.Write( simbolos[ simboloActual ]);
            Thread.Sleep(500);
            if (Console.KeyAvailable)
            {
                ConsoleKeyInfo tecla = Console.ReadKey(true);
                if (tecla.Key == ConsoleKey.RightArrow) posX++;
                if (tecla.Key == ConsoleKey.LeftArrow) posX--;
                if (tecla.Key == ConsoleKey.Escape) terminado = true;
            }
            simboloActual++;
            if (simboloActual > 3) simboloActual = 0;
        }
        while ( ! terminado );
    }
}

```

Al igual que en este ejemplo, será recomendable hacer una pequeña **pausa** entre una comprobación de teclas y la siguiente, con `Thread.Sleep`, tanto para que la animación no sea demasiado rápida como para no hacer un consumo muy alto de procesador para tareas poco importantes.

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).

Ejercicios propuestos:

(12.2.1) Crea un programa que muestre una "pelota" (la letra "O") rebotando en los bordes de la pantalla. Para que no se mueva demasiado rápido, haz una pausa de 50 milisegundos entre un "fotograma" y otro.

(12.2.2) Crea una versión de la "base de datos de ficheros" (ejemplo 04_06a) que use colores para ayudar a distinguir los mensajes del programa de las respuestas del usuario, y que no necesite pulsar Intro tras escoger cada opción.

(12.2.3) Crea un programa que permita "dibujar" en consola, moviendo el cursor con las flechas del teclado y pulsando "espacio" para dibujar un punto o borrarlo.

(12.2.4) Crea una versión del programa de "dibujar" en consola (12.2.3), que permita escribir más caracteres (por ejemplo, las letras), así como mostrar ayuda (pulsando F1), guardar el contenido de la pantalla en un fichero de texto (con F2) o recuperarlo (con F3).

(12.2.5) Crea una versión mejorada del programa 12.1.2 (mostrar el reloj actualizado en pantalla, que lo dibuje siempre en la esquina superior derecha de la pantalla).

12.3. 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 crear un directorio con:

```
// Ejemplo_12_03a.cs
// Crear un directorio
// Introducción a C#, por Nacho Cabanes

using System.IO;

public class Ejemplo_12_03a
{
    public static void Main()
    {
        string miDirectorio = @"d:\datos";
        if (!Directory.Exists(miDirectorio))
            Directory.CreateDirectory(miDirectorio);
    }
}
```

(la clase `Directory` está declarada en el espacio de nombres `System.IO`, por lo que deberemos añadirlo entre los "using" de nuestro programa).

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:

```
// Ejemplo_12_03b.cs
// Lista de ficheros en un directorio
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_12_03b
{
    public static void Main()
    {
        string miDirectorio = @"c:\";
        string[] listaFicheros;

        listaFicheros = Directory.GetFiles(miDirectorio);
        foreach(string fichero in listaFicheros)
            Console.WriteLine(fichero);
    }
}
```

Se puede añadir un segundo parámetro a "GetFiles", que sería el patrón que deben seguir los nombres de los ficheros que buscamos, como "*.txt". Por ejemplo, podríamos saber todos los fuentes de C# (*.cs) de la carpeta actual (".") con:

```
// Ejemplo_12_03c.cs
// Lista de ficheros en un directorio
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_12_03c
{
    public static void Main()
    {
        string[] listaFicheros = Directory.GetFiles(".", "*.cs");
        foreach(string fichero in listaFicheros)
            Console.WriteLine(fichero);
    }
}
```

Si necesitamos más detalles, 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:

```
// Ejemplo_12_03d.cs
// Lista detallada de ficheros en un directorio
// Introducción a C#, por Nacho Cabanes

using System;
using System.IO;

public class Ejemplo_12_03d
{
    public static void Main()
    {
        string miDirectorio = @"c:\\";
        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 6775930880, creado 15/07/2013 17:48:07

Ejercicios propuestos:

(12.3.1) Crea un programa que muestre en pantalla el contenido de un fichero de texto, cuyo nombre escoja el usuario. Si el usuario no sabe el nombre, podrá pulsar "Intro" y se le mostrará la lista de ficheros existentes en el directorio actual, para luego volver a preguntarle el nombre del fichero.

(12.3.2) Crea un programa que cree un fichero de texto a partir del contenido de todos los ficheros de texto existentes en la carpeta actual.

(12.3.3) Crea un programa que permita "pasear" por la carpeta actual, al estilo del antiguo "Comandante Norton": mostrará la lista de ficheros y subdirectorios de la carpeta actual, y permitirá al usuario moverse hacia arriba o abajo dentro de la lista usando las flechas del cursor. El elemento seleccionado se mostrará en color distinto del resto.

(12.3.4) Mejora el ejercicio 12.3.3 para que muestre directorios (en primer lugar) y ficheros (a continuación), y permita entrar a un directorio si se pulsa Intro sobre él (en ese momento, se actualizará la lista de ficheros y directorios, para mostrar el contenido del directorio al que se ha accedido).

(12.3.5) Mejora el ejercicio 12.3.4 para que contenga dos paneles, uno al lado del otro, cada uno de los cuales podrá estar mostrando el contenido de un directorio distinto. Si se pulsa el "tabulador", cambiará el panel activo.

(12.3.6) Mejora el ejercicio 12.3.5, para que se pueda "seleccionar un fichero" pulsando "Espacio" o "Insert". Los ficheros seleccionados se mostrarán en un color distinto. Se podrán deseleccionar volviendo a pulsar "Espacio" o "Insert". Si se pulsa F5, los ficheros seleccionados en la carpeta actual del panel actual se copiarán a la carpeta del otro panel. Mientras se están copiando, el programa debe mostrar una "barra de progreso" de color amarillo, que indicará el porcentaje de ficheros que ya se han copiado.

12.4. 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();
```

Un programa completo que lanzara el bloc de notas y que esperase a que se cerrase podría ser:

```
// Ejemplo_12_04a.cs
// Lanzar otro proceso y esperar
// Introducción a C#, por Nacho Cabanes

using System;
using System.Diagnostics;

public class Ejemplo_12_04a
{
    public static void Main()
    {
        Console.WriteLine("Lanzando el bloc de notas...");
```

```

        Process proc = Process.Start("notepad.exe");
        Console.WriteLine("Esperando a que se cierre...");
        proc.WaitForExit();
        Console.WriteLine("Terminado!");
    }
}

```

Ejercicios propuestos:

(12.4.1) Mejora el ejercicio 12.3.3 (la versión básica del programa "tipo Comandante Norton") para que, si se pulsa Intro sobre un cierto fichero, lance el correspondiente proceso.

(12.4.2) Aplica esta misma mejora al ejercicio 12.3.5 (la versión con dos paneles del programa "tipo Comandante Norton").

(12.4.3) Crea un programa que mida el tiempo que tarda en ejecutarse un cierto proceso. Este proceso se le indicará como parámetro en la línea de comandos.

12.5. 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:

```

// Ejemplo_12_05a.cs
// Información sobre el sistema
// Introducción a C#, por Nacho Cabanes

using System;
using System.Diagnostics;

public class Ejemplo_12_05a
{
    public static void Main()
    {
        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("Version 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));
    }
}

```

Ejercicios propuestos:

(12.5.1) Mejora el ejercicio 12.4.2 (la versión más completa del programa "tipo Comandante Norton") para que se pueda pasar a "navegar" por otra unidad de disco. Si se pulsa Alt+F1, se mostrará la lista de unidades lógicas, el usuario podrá escoger una de ellas, y esa pasará a ser la unidad activa en el panel izquierdo. Si se pulsa Alt+F2, se realizará el mismo proceso, pero cambiará la unidad activa en el panel derecho.

12.6. 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_12_06a.cs
// Ejemplo de descarga y análisis de una web:
// Muestra las líneas que contienen "Pascal"
// Introducción a C#, por Nacho Cabanes

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: | <a href="pascal/index.php">Pascal / Delphi</a>
204:    <li><a href="pascal/index.php">Pascal/Delphi</a></li>

```

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_12_06b.cs
// Ejemplo de envío y recepción de frases a través de la red
// Introducción a C#, por Nacho Cabanes

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 ComunicacionRed
{
    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.GetHostEntry(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.

Ejercicios propuestos:

(12.6.1) Crea un juego de "3 en raya" en red, para dos jugadores, en el que cada jugador escoja un movimiento, pero ambos vean un mismo estado del tablero.

(12.6.2) Crea un programa básico de chat, en el que dos usuarios puedan comunicarse, sin necesidad de seguir turnos estrictos.

(12.6.3) Crea un programa que monitorice cambios en una página web, comparando el contenido actual con una copia guardada en fichero. Deberá mostrar en pantalla un mensaje que avise al usuario de si hay cambios o no.

(12.6.4) Crea un programa que descargue todo un sitio web, partiendo de la página que indique el usuario, analizando los enlaces que contiene y descargando de forma recursiva las páginas que corresponden a dichos enlaces. Sólo deberá procesar los enlaces internos (en el mismo sitio web), no las páginas externas (alojadas en otros sitios web).

(12.6.5) Crea un juego de "barquitos" en red, para dos jugadores, en el que cada jugador decida dónde quiere colocar sus barcos, y luego cada uno de ellos "dispare" por turnos, escogiendo una casilla (por ejemplo, "B5"), y siendo avisado de si ha acertado ("impacto") o no ("agua"). Los barcos se podrán colocar en horizontal o en vertical sobre un tablero de 8x8, y serán: 4 de longitud 1, 3 de longitud 2, 2 de longitud 3 y uno de longitud 4. Cada jugador verá el estado de su

propio tablero y los impactos que ha conseguido en el tablero del otro jugador. El juego terminará cuando uno de los jugadores hunda toda la flota del otro.

(12.6.6) Mejora el juego de "barquitos" (12.6.5) para que el aviso de impacto sea más detallado ("tocado" o "hundido", según el caso).

(12.6.7) Mejora el juego de "barquitos" completo (12.6.6) para que un jugador pueda decidir aplazar la partida, y entonces el resultado quede guardado en fichero, y en la siguiente sesión se reanude el juego en el punto en el que quedó.

13. Otras características avanzadas de C#

13.1. 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 (o incluso en distintos grupos de un mismo gran proyecto) creen 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 forma que cualquier usuario de esas clase pueda simultanear su uso con el de las incluidas en la plataforma .Net.

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.Write("Press any key to continue . . . ");
            Console.ReadKey(true);
        }
    }
}

```

Nosotros vamos a preparar 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_13_01a.cs
// Ejemplo de uso de "namespaces"
// Introducción a C#, por Nacho Cabanes

using System;

namespace ConsolaAmpliada
{
    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");
        ConsolaAmpliada.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 "ConsolaAmpliada.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

13.2. 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	~	~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 = 1111
Suma exclusiva (xor)	1 sólo si los 2 bits son distintos	^	1101 ^ 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_13_02a.cs
// Operaciones a nivel de bits
// Introducción a C#, por 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
 Desplacemos a a la izquierda: 134
 Desplacemos 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: $1^1 = 0$, $1^0 = 1$, $0^0 = 0$

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;
...
```

Ejercicios propuestos

(13.2.1) Crea un programa que lea un fichero de texto, encripte cada línea haciendo un XOR de los caracteres que la forman con un cierto dato prefijado (y pequeño, como 3, por ejemplo) y vuelque el resultado a un nuevo fichero.

13.3. 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_13_03a.cs
// Ejemplo de enumeraciones
// Introducción a C#, por 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`).

Ejercicios propuestos

(13.3.1) Crea una versión de la base de datos de ficheros con colores (ejercicio 12.2.2) en la que las opciones sean parte de una enumeración.

13.4. 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_13_04a.cs
// Ejemplo de propiedades (1)
// Introducción a C#, por 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 ocurriría 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".

Ejercicios propuestos

(13.4.1) Crea una nueva versión del ejercicio de la clase Persona (6.7.1), en la que el "nombre" sea una propiedad, con sus correspondientes "get" y "set".

13.5. Introducción a las expresiones regulares.

Las "expresiones regulares" permiten hacer comparaciones mucho más abstractas que si se usa un simple "Contains". 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: una 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:

```
// Ejemplo_13_05a.cs
// Ejemplo de expresiones regulares
// Introducción a C#, por Nacho Cabanes

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)

Ejercicios propuestos

(13.5.1) Crea una función que valide códigos postales españoles (5 cifras numéricas): devolverá true si la cadena recibida como parámetro es un código postal válido.

(13.5.2) Crea una función que diga si una cadena que se le pase como parámetro parece un correo electrónico válido.

13.6. 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_13_06a.cs
// Operador coma
// Introducción a C#, por Nacho Cabanes

using System;

public class OperadorComa
{
    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 {1}", i, j);
    }
}
```

Vamos a ver qué hace este "for":

- Los valores iniciales son i=0, j=1.
- Se repetirá mientras que i <= 5 y j <= 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 1
i vale 1 y j vale 3
i vale 2 y j vale 5
i vale 3 y j vale 7
i vale 4 y j vale 9
i vale 5 y j vale 11
```

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)
```

13.7. Variables con tipo implícito

A partir de Visual C# 3.0, existe la posibilidad de no declarar de forma explícita el tipo de una variable, sino que sea el propio compilador el que lo deduzca del contexto. Para ello, se utiliza la palabra "var", como en este ejemplo:

```
// Ejemplo_13_07a.cs
// Ejemplo de uso de "var"
// Introducción a C#, por Nacho Cabanes
```

```

using System;

public class UsoVar
{
    public static void Main()
    {
        var n = 5;
        Console.WriteLine("n vale {0} y es de tipo {1}",
            n, n.GetType());

        var condicion = 5 == 7;
        Console.WriteLine("condicion vale {0} y es de tipo {1}",
            condicion, condicion.GetType());

        var letra = 'a';
        Console.WriteLine("letra vale {0} y es de tipo {1}",
            letra, letra.GetType());

        var pi = 3.1416;
        Console.WriteLine("pi vale {0} y es de tipo {1}",
            pi, pi.GetType());

        var texto = "Hola";
        Console.WriteLine("texto vale {0} y es de tipo {1}",
            texto, texto.GetType());
    }
}

```

Como se ve en este ejemplo, si necesitáramos saber de qué tipo es una variable (lo que no es habitual, porque si se usa "var" es para despreocuparnos de esos detalles), lo podríamos conseguir con "GetType()".

Ejercicios propuestos

(13.7.1) Crea un programa que pida al usuario una cantidad de kilómetros y muestre su equivalencia en millas. El valor de conversión debe estar en una variable definida con "var".

(13.7.2) Crea un programa que muestre la primera línea de un fichero de texto. Tanto el fichero como la línea se deben declarar con "var".

13.8. Contacto con LINQ

LINQ (Language-integrated Query, consulta integrada en el lenguaje) es una posibilidad añadida en la plataforma .Net versión 3.5, y accesible a partir de Visual Studio 2008, que nos permite hacer consultas a cualquier colección de datos usando una sintaxis que recuerda a la de SQL (aunque se escribe "casi al revés": primero el conjunto de datos, luego la condición y luego la variable a devolver).

Por ejemplo, podemos obtener los números enteros de un array cuyo valor es mayor que 10 con:

```
// Ejemplo_13_08a.cs
// Ejemplo básico de LINQ
// Introducción a C#, por Nacho Cabanes

using System;
using System.Linq;

public class EjemploLinq1
{
    public static void Main()
    {
        int[] datos = { 20, 5, 7, 4, 25, 18, 5, 8, 21, 2 };

        var result =
            from n in datos
            where n > 10
            select n;

        foreach(int i in result)
            Console.Write("{0} ", i);
        Console.WriteLine();
    }
}
```

También podemos recorrer la información de una lista de cadenas de texto y obtenerla ordenada, así:

```
// Ejemplo_13_08b.cs
// Segundo ejemplo de LINQ
// Introducción a C#, por Nacho Cabanes

using System;
using System.Linq;
using System.Collections.Generic;

public class EjemploLinq2
{
    public static void Main()
    {
        List<string> nombres = new List<string>
            { "pan", "carne", "queso", "manzana", "natillas" };

        var resultado =
            from nombre in nombres
            orderby nombre
            select nombre;

        foreach (string n in resultado)
            Console.Write("{0} ", n);
        Console.WriteLine();
    }
}
```

Profundizar en LINQ es algo que queda fuera del propósito de este texto. Si quieres saber más, puedes acudir a la propia referencia oficial en línea, en MSDN:

<https://msdn.microsoft.com/es-es/library/bb397926.aspx>

Ejercicios propuestos

(13.8.1) Crea un programa que pida al usuario varios números enteros, los guarde en una lista y luego muestre todos los que sean positivos, ordenados, empleando LINQ.

(13.8.2) Crea un programa que prepare un array de palabras. Luego, el usuario debe introducir una palabra, y el programa responderá si es palabra está en el array o no, utilizando LINQ.

13.9. Lo que no vamos a ver...

En C# (y la plataforma "punto net") 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).
- Funciones lambda.
- El preprocesador.
- Entornos gráficos (Windows Forms, WPF).
- Uso en servidores Web (ASP.Net).
- ...

14. Depuración, prueba y documentación de programas

14.1. 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, pero las diferencias con versiones posteriores de Visual Studio deberían ser mínimas.

14.2. Depurando desde Visual Studio

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;

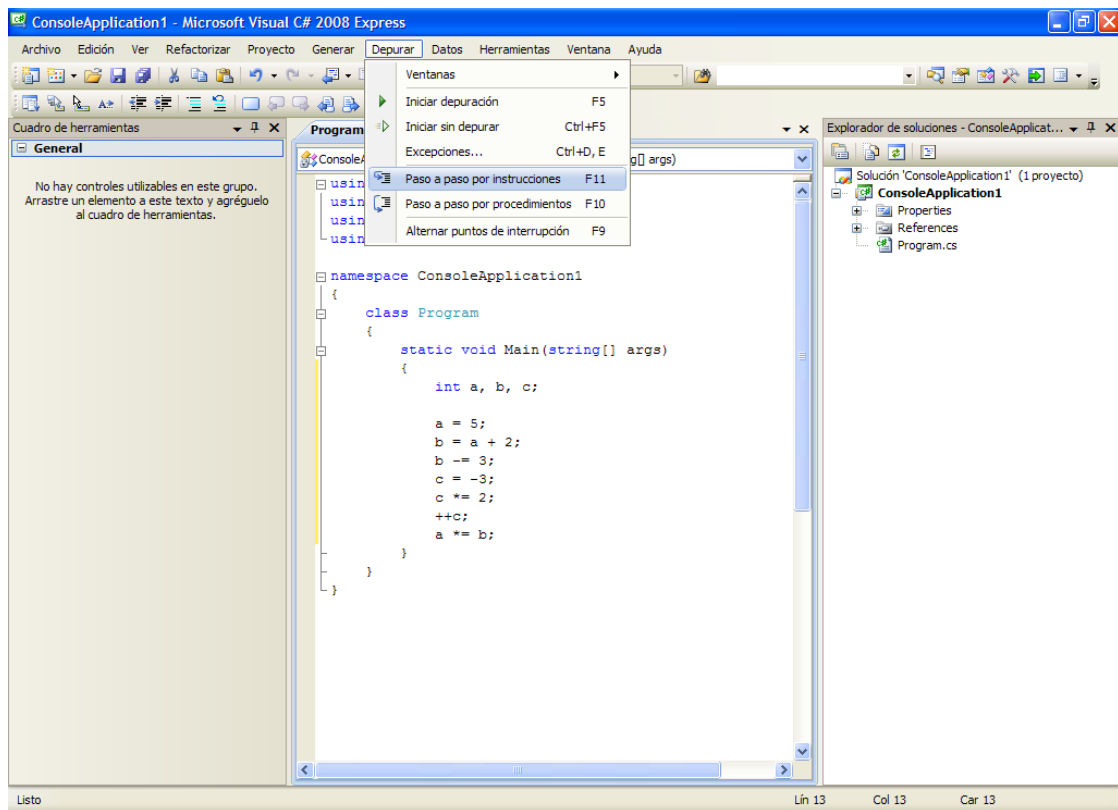
namespace EjemploDeOperaciones
{
    class Program
    {
        static void Main(string[] args)
        {
            int a, b, c;

            a = 5;
            b = a + 2;
            b -= 3;
            c = -3;
            c *= 2;
            ++c;
            a *= b;
        }
    }
}
```

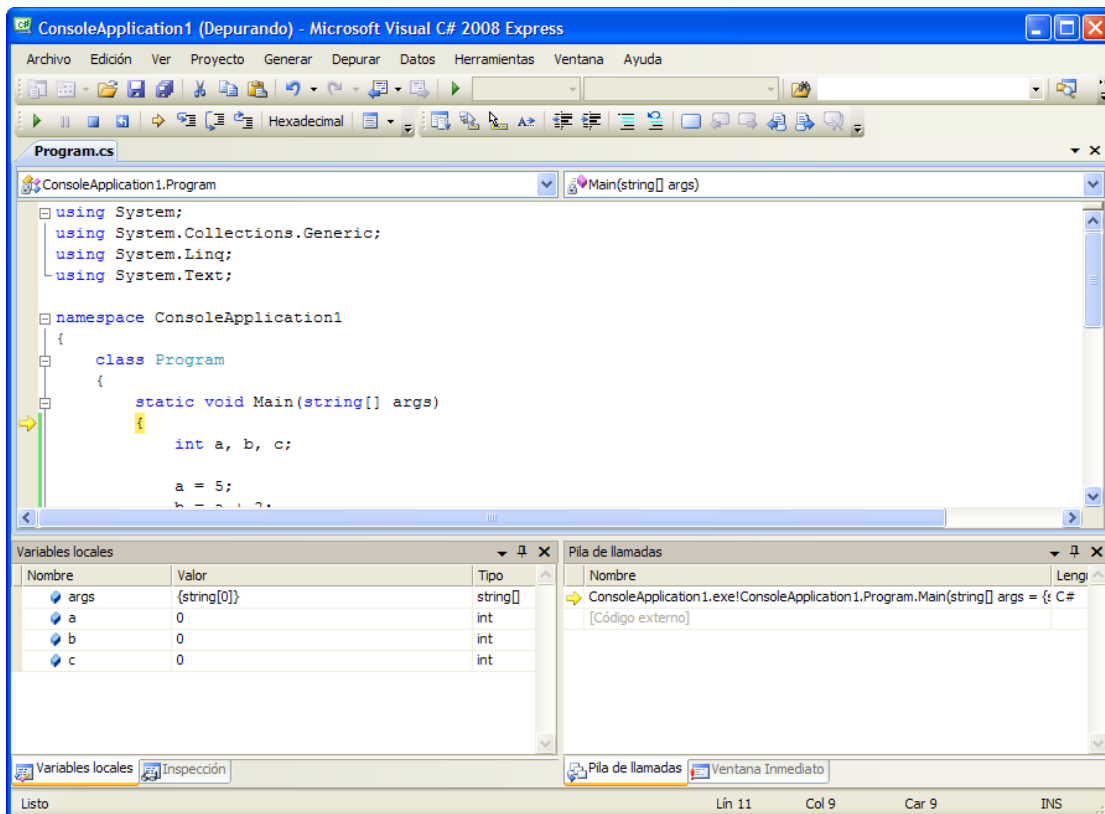
}

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.

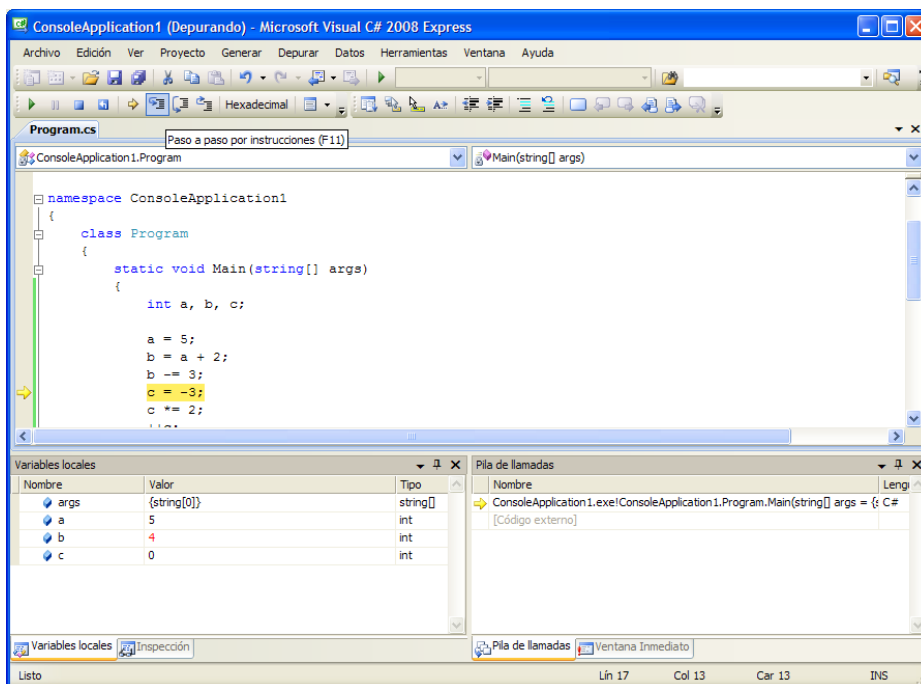
Para avanzar paso a paso y ver los valores de las variables, entramos al menú "Depurar". En él aparece la opción "Paso a paso por instrucciones" (al que corresponde la tecla F11):



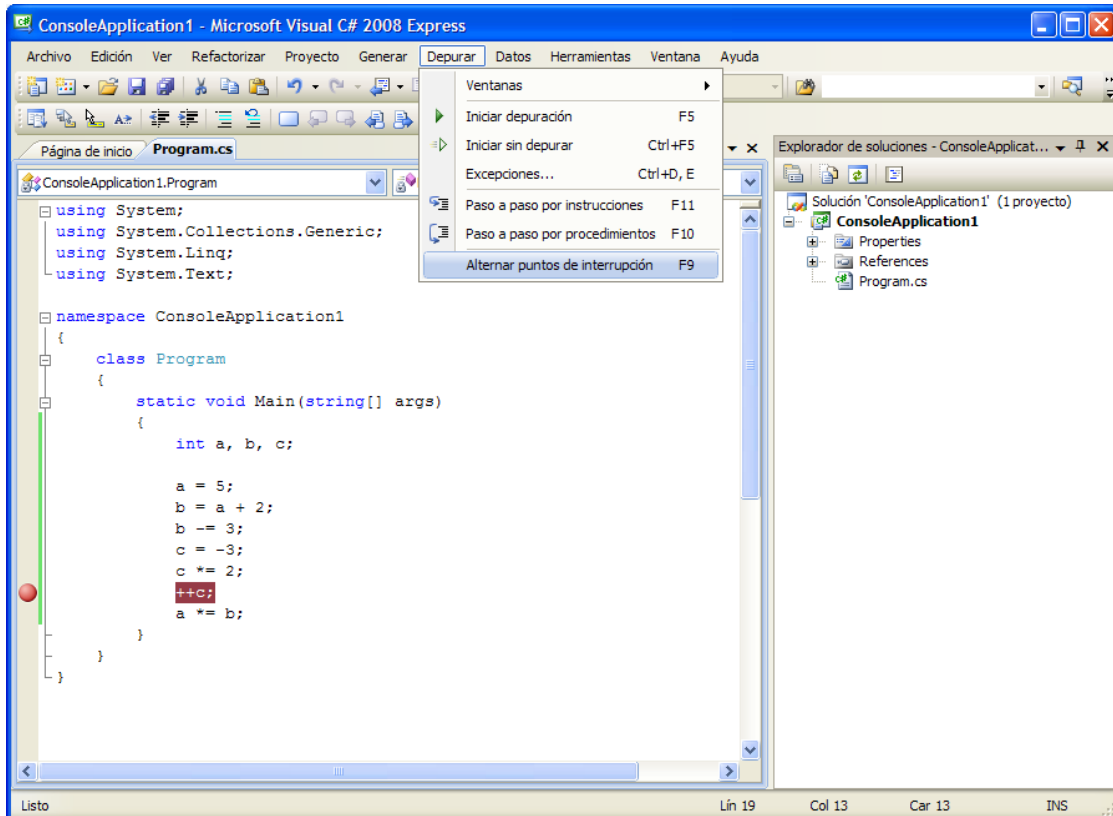
Si escogemos esa opción del menú o pulsamos F11, aparece una ventana inferior con la lista de variables, y un nuevo cursor amarillo, que se queda al principio del programa:



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 (el cambio más reciente se verá en color rojo), 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 como alternativa, podemos hacer clic con el ratón en el margen izquierdo del programa, junto a esa línea):



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.

14.3. 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:

$$x^2 - 1 = 0 \rightarrow x_1 = 1, x_2 = -1$$

$$x^2 = 0 \rightarrow x_1 = 0, x_2 = \text{No existe (solución única)}$$

$$x^2 - 3x = 0 \rightarrow x_1 = 3, x_2 = 0$$

$$2x^2 - 2 = 0 \rightarrow x_1 = 1, x_2 = -1$$

Estos casos de prueba se podrían convertir en un programa como:

```
// Ejemplo_14_03a.cs
// Ejemplo de realización de pruebas
// Introducción a C#, por Nacho Cabanes

using System;

public class SegundoGrado
{
    public 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);
        }
    }
}

public class PruebaSegundoGrado
{
    public static void Main()
    {
        float soluc1, soluc2;

        Console.WriteLine("Probando ecuaciones de segundo grado");

        SegundoGrado ecuacion = new SegundoGrado();

        Console.WriteLine("Probando x2 - 1 = 0 ...");
        ecuacion.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 ...");
ecuacion.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 ...");
ecuacion.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 ...");
ecuacion.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 $(2 \cdot a)$, de modo que estamos dividiendo entre 2 y luego multiplicando por a.

La mayoría de lenguajes modernos tienen funcionalidades de creación de pruebas ya incorporadas como parte del lenguaje, normalmente en forma de una función llamada "**Assert**". En C#, se trata de `Debug.Assert`", que es parte de `System.Diagnostics`, de modo que el programa quedaría así:

```

// Ejemplo_14_03b.cs
// Segundo ejemplo de realización de pruebas, con Assert
// Introducción a C#, por Nacho Cabanes

```

```

using System;
using System.Diagnostics;

public class SegundoGrado
{
    public 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);
        }
    }
}

public class PruebaSegundoGrado
{
    public static void Main()
    {
        float soluc1, soluc2;

        Console.WriteLine("Probando ecuaciones de segundo grado");

        SegundoGrado ecuacion = new SegundoGrado();

        Console.WriteLine("Probando x2 - 1 = 0 ...");
        ecuacion.Resolver((float)1, (float)0, (float)-1,
            out soluc1, out soluc2);
        Debug.Assert((soluc1 == 1) && (soluc2 == -1));

        Console.WriteLine("Probando x2 = 0 ...");
        ecuacion.Resolver((float)1, (float)0, (float)0,
            out soluc1, out soluc2);
        Debug.Assert((soluc1 == 0) && (soluc2 == -9999));

        Console.WriteLine("Probando x2 -3x = 0 ...");
        ecuacion.Resolver((float)1, (float)-3, (float)0,
            out soluc1, out soluc2);
        Debug.Assert((soluc1 == 3) && (soluc2 == 0));

        Console.WriteLine("Probando 2x2 - 2 = 0 ...");
    }
}

```



```

    ecuacion.Resolver((float)2, (float)0, (float)-2,
        out soluc1, out soluc2);
    Debug.Assert((soluc1 == 1) && (soluc2 == -1));
}
}

```

A lanzar este programa, se detendrá la ejecución y mostrará una ventana de aviso con el texto "Error de aserción" al llegar a la línea 67, con botones para Salir, Depurar o Continuar ignorando el error.

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".

14.4. 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.

14.4.1. 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-codigo.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.

14.4.2. 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#:

[Página principal](#) [Clases](#)

[Lista de clases](#) [Índice de clases](#) [Jerarquía de la clase](#) [Miembros de las clases](#)

Referencia de la Clase Personaje

Diagrama de herencias de Personaje

```

classDiagram
    class ElemGrafico
    class Personaje
    Personaje --|> ElemGrafico
  
```

[Lista de todos los miembros.](#)

Métodos públicos

Personaje (Juego j)
void MoverDerecha ()
void MoverIzquierda ()
void MoverArriba ()
void MoverAbajo ()
int GetVidas ()
void SetVidas (int n)
void QuitarVida ()
void Morir ()
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 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`.

La forma que propone Microsoft de crear comentarios de documentación en C# es usar etiquetas XML, así:

```

    /// <summary>Mueve el personaje a la derecha, si es posible</summary>
    /// <seealso cref="MoverIzquierda"/>
    public void MoverDerecha()
    {
        (...)
    }
}

```

Las etiquetas que sugiere Microsoft son <c> <para> <see> <code> <param> <seealso> <example> <paramref> <summary> <exception> <permission> <typeparam> <include>* <remarks> <typeparamref> <list> <returns> <value>.

Puedes leer más sobre el uso que se sugiere para cada una de ellas aquí:

<https://msdn.microsoft.com/en-us/library/5ast78ax.aspx>

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 abreviar **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 $1 \text{ Kb} = 1024 \text{ bytes}$.

Los Kb eran unidades típicas para medir la memoria de ordenadores: 640 Kb fue mucho tiempo la memoria habitual en los primeros IBM PC y equipos similares. Por otra parte, una página mecanografiada suele ocupar entre 2 K (cerca de 2000 letras) y 4 Kb.

Cuando se manejan datos más extensos, necesitamos otro múltiplo, el **megabyte** o **Mb**, que es 1000 Kb (en realidad 1024 Kb) o algo más de un millón de bytes. Por ejemplo, en un diskette cabían 1.44 Mb, y en un Compact Disc para ordenador (CD-ROM) se pueden almacenar hasta 700 Mb.

Para unidades de almacenamiento de gran capacidad, su tamaño no se suele medir en megabytes, sino en el múltiplo siguiente: en **gigabytes**, con la correspondencia $1 \text{ Gb} = 1024 \text{ Mb}$. Así, es habitual que un equipo actual tenga una memoria RAM entre 4 y 8 Gb, así como un disco duro de entre 500 y 2.000 Mb (2 **Terabytes**).

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:

(Ap1.1.1) ¿Cuántas letras se podrían almacenar en una agenda electrónica que tenga 32 Kb de capacidad?

(Ap1.1.2) 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?

(Ap1.1.3) ¿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?

(Ap1.1.4) ¿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 encontremos la secuencia 00000010 la interpretaremos como una letra A, y la combinación 00000011 como una letra B, y así sucesivamente (aunque existe un estándar distinto, que comentaremos un poco más adelante).

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 66, 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). En el próximo apéndice tendrás un extracto de este código.

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. Por eso, existen estándares más modernos, como UTF-8, que comentaremos en el siguiente apartado.

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**.

Ejercicios propuestos:

(Ap1.2.1) Un número entero largo de 64 bits, ¿cuántos bytes ocupa?

(Ap1.2.2) En una conexión de red de 100 Mb/s, ¿cuánto tiempo tardaría en enviar 630 Mbytes de datos?

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	i	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 UTF-8, 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.

Ejercicios propuestos:

(Ap2.1) Crea un programa en C# que muestre una tabla ASCII básica, desde el carácter 32 hasta el 127, en filas de 16 caracteres cada una.

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:

$$254 = 3 \cdot 10^3 + 2 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$$

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:

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

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 (resto: 109)
109 / 64 = 1 (resto: 45)
45 / 32 = 1 (resto: 13)
13 / 16 = 0 (resto: 13)
13 / 8 = 1 (resto: 5)
5 / 4 = 1 (resto: 1)
1 / 2 = 0 (resto: 1)
1 / 1 = 1 (hemos terminado).
```

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:

$0 \cdot 0 = 0$	$0 \cdot 1 = 0$	$1 \cdot 0 = 0$	$1 \cdot 1 = 1$	
$0 + 0 = 0$	$0 + 1 = 1$	$1 + 0 = 1$	$1 + 1 = 10$	(en decimal: 2)

Ejercicios propuestos:

(Ap3.1.1) Expresa en sistema binario los números decimales 17, 101, 83, 45.

(Ap3.1.2) Expresa en sistema decimal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101

(Ap3.1.3) Suma los números 01100110+10110010, 11111111+00101101. Comprueba el resultado sumando los números decimales obtenidos en el ejercicio anterior.

(Ap3.1.4) Multiplica los números binarios de 4 bits 0100·1011, 1001·0011. Comprueba el resultado convirtiéndolos a decimal.

(Ap3.1.5) Crea un programa en C# que convierta a binario los números (en base 10) que introduzca el usuario.

(Ap3.1.6) Crea un programa en C# que convierta a base 10 los números binarios que introduzca el usuario.

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:

```
254 (decimal) ->
254 / 64 = 3 (resto: 62)
62 / 8 = 7 (resto: 6)
6 / 1 = 6 (se terminó)
```

de modo que

$$254 = 3 \cdot 8^2 + 7 \cdot 8^1 + 6 \cdot 8^0$$

o bien

254 (decimal) = 376 (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 (decimal) = 011 111 110 (binario)

011 (binario) = 3 (decimal y octal)

111 (binario) = 7 (decimal y octal)

110 (binario) = 6 (decimal y octal)

de modo que

254 (decimal) = 011 111 110 (binario) = 376 (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 (octal) = $4 \cdot 64 + 2 \cdot 8 + 3 \cdot 1 = 275$ (decimal)

De cualquier modo, el sistema octal no es el que más se utiliza en la práctica, sino el hexadecimal...

Ejercicios propuestos:

(Ap3.2.1) Expresar en sistema octal los números decimales 17, 101, 83, 45.

(Ap3.2.2) Expresar en sistema octal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101

(Ap3.2.3) Expresar en el sistema binario los números octales 171, 243, 105, 45.

(Ap3.2.4) Expresar en el sistema decimal los números octales 162, 76, 241, 102.

(Ap3.1.5) Crea un programa en C# que convierta a octal los números (en base 10) que introduzca el usuario.

(Ap3.1.6) Crea un programa en C# que convierta a base 10 los números octales que introduzca el usuario.

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 (decimal) = FE (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 (decimal) = D43B (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:

```
110010100100100101010100111 =>
0110 0101 0010 0100 1010 1010 0111 = 6524AA7
```

Ejercicios propuestos:

(Ap3.3.1) Expresa en sistema hexadecimal los números decimales 18, 131, 83, 245.

(Ap3.3.2) Expresa en sistema hexadecimal los números binarios de 8 bits: 01100110, 10110010, 11111111, 00101101

(Ap3.3.3) Expresa en el sistema binario los números hexadecimales 2F, 37, A0, 1A2.

(Ap3.3.4) Expresa en el sistema decimal los números hexadecimales 1B2, 76, E1, 2A.

(Ap3.3.5) Crea un programa en C# que convierta a hexadecimal los números (en base 10) que introduzca el usuario.

(Ap3.3.6) Crea un programa en C# que convierta a base 10 los números hexadecimales que introduzca el usuario.

(Ap3.3.7) Crea un programa en C# que convierta a hexadecimal los números binarios que introduzca el usuario.

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 (decimal) = 0011	-3 = 1011
6 (decimal) = 0110	-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 (decimal) = 0000 1101 - 13 (decimal) = 1000 1101
34 (decimal) = 0010 0010 - 34 (decimal) = 1010 0010
13 + 34 = 0000 1101 + 0010 0010 = 0010 1111 = 47 (correcto)
(-13) + (-34) = 1000 1101 + 1010 0010 = 0010 1111 = 47 (INCORRECTO)
13 + (-34) = 0000 1101 + 1010 0010 = 1010 1111 = -47 (INCORRECTO)
```

Complemento a 1: se cambian los ceros por unos para expresar los números negativos.

Por ejemplo, con 4 bits

```
3 (decimal) = 0011      -3 = 1100
6 (decimal) = 0110      -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: convierte los números decimales 13, 34, -13, -34 a sistema binario, usando complemento a uno para expresar los números negativos. Calcula (en binario) el resultado de las operaciones 13+34, (-13)+(-34), 13+(-34) y comprueba 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 (decimal) = 0011      -3 = 1101
6 (decimal) = 0110      -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 siempre se realizan de forma correcta, y no tiene problemas para expresar el número 0 (que siempre se almacena como 00000000).

Ejercicio propuesto (Ap3.4.1): Convierte los números decimales 13, 34, -13, -34 a sistema binario, usando complemento a dos para expresar los números negativos. Calcula (en binario) el resultado de las operaciones $13+34$, $(-13)+(-34)$, $13+(-34)$ y comprueba 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 "ushort" (entero corto sin signo) en C# podría tomar valores entre 0 y 65.535, mientras que un "short" (entero corto con signo) podría tomar valores entre +32.767 y -32.768.

Apéndice 4. SDL

Ap4.1. 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/gamepad, así como reproducir sonidos.

Tao.SDL es una adaptación de esta librería, que permite emplearla desde C#. Las primeras versiones se pueden descargar desde <http://www.mono-project.com/Tao> y las versiones más recientes desde <http://sourceforge.net/projects/taoframework/>

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 haremos un poco más adelante.

Ap4.2. 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.

```
// sdl01.cs
// Primer acercamiento a SDL
// Introducción a C#, por Nacho Cabanes

using Tao.Sdl;
using System; // Para IntPtr (puntero: imágenes, etc)

public class Sdl01
{
    public static void Main()
    {
        short anchoPantalla = 800;
        short altoPantalla = 600;
        int bitsColor = 24;
        int flags = Sdl.SDL_HWSURFACE | Sdl.SDL_DOUBLEBUF | Sdl.SDL_ANYFORMAT;
        IntPtr pantallaOculta;

        // Inicializamos SDL
        Sdl.SDL_Init(Sdl.SDL_INIT_EVERYTHING);
```



```

pantallaOculta = Sdl.SDL_SetVideoMode(
    anchoPantalla,
    altoPantalla,
    bitsColor,
    flags);

// 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
IntPtr imagen;
imagen = Sdl.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;
Sdl.SDL_Rect 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, pantallaOculta, ref dest);

// Mostramos la pantalla oculta
Sdl.SDL_Flip(pantallaOculta);

// Y esperamos 5 segundos
System.Threading.Thread.Sleep( 5000 );

// Finalmente, cerramos SDL
Sdl.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 damos 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:

```
gmcs sdl01.cs /r:Tao.Sdl.dll /platform:x86
```

Y si empleamos Visual Studio o SharpDevelop, tendremos que:

- Crear un proyecto de "aplicación de consola".
- Reemplazar nuestro programa principal por éste.
- 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").
- Añadir el fichero Tao.Sdl.Dll a las referencias del proyecto (normalmente, pulsando el botón derecho del ratón en el apartado "Referencias" de la ventana del Explorador de soluciones y escogiendo la opción "Agregar referencia").
- Compilar y probar.
- Cuando todo funcione correctamente, podremos cambiar el "tipo de proyecto" (desde el menú "Proyecto", en la opción "Propiedades") a "Aplicación de Windows", para que no aparezca la ventana negra de consola cuando lanzamos nuestro programa.

No necesitaremos ninguna DLL adicional 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, rectángulos, círculos, etc).

Puedes descargar este fuente, incluyendo el proyecto de Visual Studio 2010, los ficheros DLL necesarios y una imagen de ejemplo, desde:

<http://www.nachocabanes.com/fich/descargar.php?nombre=SdlBasico.zip>

Ejercicios propuestos

(Ap4.2.1) Crea una animación con al menos 4 fotogramas, utilizando varias imágenes que se mostrarán de forma repetitiva (por ejemplo, 10 veces, con medio segundo de pausa entre una imagen y la siguiente).

Ap4.3. 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:

```
// sdl02.cs
// Segundo acercamiento a SDL
// Introducción a C#, por Nacho Cabanes

using Tao.Sdl;
using System; // Para IntPtr (puntero: imágenes, etc)

public class Sdl02
{
    public static void Main()
    {
        short anchoPantalla = 800;
        short altoPantalla = 600;
        int bitsColor = 24;
        int flags = Sdl.SDL_HWSURFACE | Sdl.SDL_DOUBLEBUF | Sdl.SDL_ANYFORMAT
            | Sdl.SDL_FULLSCREEN;
        IntPtr pantallaOculta;

        // Inicializamos SDL
        Sdl.SDL_Init(Sdl.SDL_INIT_EVERYTHING);
        pantallaOculta = Sdl.SDL_SetVideoMode(
            anchoPantalla,
            altoPantalla,
            bitsColor,
            flags);

        // 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
IntPtr imagen;
imagen = Sdl.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;
Sdl.SDL_Event suceso;
int numkeys;
byte[] teclas;

do
{
    // 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;

    // Borramos pantalla
    Sdl.SDL_Rect origen = new Sdl.SDL_Rect(0,0,
        anchoPantalla,altoPantalla);
    Sdl.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,
        pantallaOculta, ref dest);

    // Mostramos la pantalla oculta
    Sdl.SDL_Flip(pantallaOculta);

    // 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.

Ejercicios propuestos

(Ap4.3.1) Amplía el ejemplo 2 de SDL (`sdl02.cs`) para que al moverse alterne entre (al menos) dos imágenes, para dar una impresión de movimiento más real.

Ap4.4. Simplificando con clases auxiliares

Podemos simplificar un poco la estructura del programa si creamos unas clases auxiliares que nos permitan ocultar los detalles de SDL y hacer un Main más legible:

Por ejemplo, podemos buscar que el programa anterior quede simplemente así:

```
// sdl03.cs
// Tercer acercamiento a SDL: clases auxiliares
// Introducción a C#, por Nacho Cabanes

public class Sdl03
{
    public static void Main()
    {
        bool pantallaCompleta = true;
        Hardware h = new Hardware(800, 600, 24, pantallaCompleta);
        Imagen img = new Imagen("personaje.bmp", 50, 50);

        // Parte repetitiva
        bool terminado = false;
        short x = 400;
        short y = 300;
```

```

do
{
    // Miramos si se ha pulsado alguna flecha del cursor
    if (h.TeclaPulsada(Hardware.TECLA_ARR))
        y -= 2;
    if (h.TeclaPulsada(Hardware.TECLA_ABA))
        y += 2;
    if (h.TeclaPulsada(Hardware.TECLA_IZQ))
        x -= 2;
    if (h.TeclaPulsada(Hardware.TECLA_DER))
        x += 2;
    if (h.TeclaPulsada(Hardware.TECLA_ESC))
        terminado = true;
    img.MoverA(x, y);

    // Dibujar en pantalla
    h.BorrarPantallaOculta();
    h.DibujarImagenOculta(img);
    h.VisualizarOculta();

    // Y esperamos 20 ms
    h.Pausa(20);
} while (!terminado);
}
}

```

Para eso necesitaríamos una clase "Hardware", que oculte los detalles del acceso a la pantalla y al teclado. Podría ser así:

```

// hardware.cs
// Clases auxiliar Hardware para SDL
// Introducción a C#, por Nacho Cabanes

using Tao.Sdl; // Para acceder a SDL
using System; // Para IntPtr
using System.Threading; // Para Pausa (Thread.Sleep)

public class Hardware
{
    short anchoPantalla = 800;
    short altoPantalla = 600;
    int bitsColor = 24;

    IntPtr pantallaOculta;

    public Hardware(int ancho, int alto, int colores, bool pantallaCompleta)
    {
        int flags = Sdl.SDL_HWSURFACE | Sdl.SDL_DOUBLEBUF |
            Sdl.SDL_ANYFORMAT;
        if (pantallaCompleta)
            flags = flags | Sdl.SDL_FULLSCREEN;

        // Inicializamos SDL
        Sdl.SDL_Init(Sdl.SDL_INIT_EVERYTHING);
        pantallaOculta = Sdl.SDL_SetVideoMode(
            anchoPantalla,

```

```

        altoPantalla,
        bitsColor,
        flags);

    // 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);
}

~Hardware()
{
    Sdl.SDL_Quit();
}

public void BorrarPantallaOculta()
{
    Sdl.SDL_Rect origen = new Sdl.SDL_Rect(0, 0,
        anchoPantalla, altoPantalla);
    Sdl.SDL_FillRect(pantallaOculta, ref origen, 0);
}

public void DibujarImagenOculta(Imagen imagen)
{
    Sdl.SDL_Rect origen = new Sdl.SDL_Rect(0, 0,
        anchoPantalla, altoPantalla);
    Sdl.SDL_Rect dest = new Sdl.SDL_Rect(
        imagen.GetX(), imagen.GetY(),
        imagen.GetAncho(), imagen.GetAlto());
    Sdl.SDL_BlitSurface(imagen.GetPuntero(), ref origen,
        pantallaOculta, ref dest);
}

public void VisualizarOculta()
{
    Sdl.SDL_Flip(pantallaOculta);
}

public bool TeclaPulsada(int c)
{
    bool pulsada = false;
    Sdl.SDL_PumpEvents();
    Sdl.SDL_Event suceso;
    Sdl.SDL_PollEvent(out suceso);
    int numkeys;
    byte[] teclas = Tao.Sdl.Sdl.SDL_GetKeyState(out numkeys);
    if (teclas[c] == 1)
        pulsada = true;
    return pulsada;
}

public void Pausa(int milisegundos)
{
    Thread.Sleep(milisegundos);
}

// Definiciones de teclas
public static int TECLA_ESC = Sdl.SDLK_ESCAPE;
public static int TECLA_ARR = Sdl.SDLK_UP;
public static int TECLA_ABA = Sdl.SDLK_DOWN;

```

```

    public static int TECLA_DER = Sdl.SDLK_RIGHT;
    public static int TECLA_IZQ = Sdl.SDLK_LEFT;
}

```

Y una clase "Imagen" podría ocultar ese "IntPtr", además de añadirle datos adicionales, como la posición en la que se encuentra esa imagen, y otros que más adelante nos resultarán útiles, como el ancho y el alto (que nos servirán para comprobar colisiones):

```

// imagen.cs
// Clases auxiliar Imagen para SDL
// Introducción a C#, por Nacho Cabanes

using Tao.Sdl; // Para acceder a SDL
using System; // Para IntPtr

public class Imagen
{
    IntPtr imagen;
    int x, y;
    int ancho, alto;

    public Imagen(string nombreFichero, int ancho, int alto)
    {
        imagen = Sdl.SDL_LoadBMP(nombreFichero);
        if (imagen == IntPtr.Zero)
        {
            System.Console.WriteLine("Imagen inexistente!");
            Environment.Exit(1);
        }
        this.ancho = ancho;
        this.alto = alto;
    }

    public void MoverA(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public IntPtr GetPuntero() { return imagen; }
    public short GetX() { return (short)x; }
    public short GetY() { return (short)y; }
    public short GetAncho() { return (short)ancho; }
    public short GetAlto() { return (short)alto; }
}

```

Al igual que antes, al crear el proyecto deberemos añadir el fichero Tao.Sdl.Dll a sus referencias, y tendremos que copiar en la carpeta de ejecutables los ficheros Tao.Sdl.Dll, SDL.Dll y la imagen "personaje.bmp".

Ejercicios propuestos

(Ap4.4.1) Descompón en clases el ejercicio Ap4.3.1.

Ap4.5. Un fuente más modular: el "bucle de juego"

A medida que nuestro juego se va complicando, el fuente puede irse haciendo más 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", "DibujarImágenes", 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.

Podemos descomponer en funciones nuestro juego y, de paso, añadir algún otro elemento, como un premio que recoger y un enemigo que se mueve:

```
// sdl04.cs
// Cuarto acercamiento a SDL: bucle de juego
// Introducción a C#, por Nacho Cabanes

using Tao.Sdl;
using System; // Para IntPtr (puntero: imágenes, etc)

public class Juego
{
    bool pantallaCompleta = true;
    Hardware h;
    Imagen img;
    bool terminado = false;
    short x, y;

    void inicializar()
    {
        h = new Hardware(800, 600, 24, pantallaCompleta);
        img = new Imagen("personaje.bmp", 50, 50);
        x = 400;
        y = 300;
    }
}
```

```

void comprobarTeclas()
{
    if (h.TeclaPulsada(Hardware.TECLA_ARR))
        y -= 2;
    if (h.TeclaPulsada(Hardware.TECLA_ABA))
        y += 2;
    if (h.TeclaPulsada(Hardware.TECLA_IZQ))
        x -= 2;
    if (h.TeclaPulsada(Hardware.TECLA_DER))
        x += 2;
    if (h.TeclaPulsada(Hardware.TECLA_ESC))
        terminado = true;
    img.MoverA(x, y);
}

void comprobarColisiones()
{
    // Todavía no comprobamos colisiones con enemigos
    // ni con obstáculos
}

void moverElementos()
{
    // Todavía no hay ningún elemento que se mueva solo
}

void dibujarElementos()
{
    h.BorrarPantallaOculta();
    h.DibujarImagenOculta(img);
    h.VisualizarOculta();
}

void pausaFotograma()
{
    // Esperamos 20 ms
    h.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());
}
}

```

(Las clases Hardware e Imagen no cambian).

Ejercicios propuestos

(Ap4.5.1) Refina el ejercicio Ap4.4.1, para que tenga un fuente modular, siguiendo un bucle de juego clásico.

Ap4.6. 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.

También tenemos que inicializar SdlTtf después de la inicialización básica de SDL (al final del método Inicializar de la clase Hardware):

```
SdlTtf.TTF_Init();
```

El tipo de letra estará accesible a través de un puntero genérico "IntPtr", igual que ocurría con las imágenes, por lo que, igual que hicimos con éstas, podemos crear una clase auxiliar, llamada "Fuente":

```

// fuente.cs
// Clases auxiliar Fuente para SDL
// Introducción a C#, por Nacho Cabanes

using Tao.Sdl; // Para acceder a SDL
using System; // Para IntPtr

public class Fuente
{
    IntPtr tipoDeLetra;

    public Fuente(string nombreFichero, short tamanyo)
    {
        Cargar(nombreFichero, tamanyo);
    }

    public void Cargar(string nombreFichero, short tamanyo)
    {

```

```

        tipoDeLetra = SdlTtf.TTF_OpenFont(nombreFichero, tamanyo);
        if (tipoDeLetra == IntPtr.Zero)
        {
            System.Console.WriteLine("Tipo de letra inexistente!");
            Environment.Exit(2);
        }
    }

    public IntPtr GetPuntero()
    {
        return tipoDeLetra;
    }
}

```

Y en la clase Hardware añadiremos un método `EscribirTextoOculta`, que dé por nosotros los pasos que son necesarios en SDL: crear una imagen a partir del texto, incluirla dentro de un rectángulo y volcarlo a otro rectángulo en la pantalla oculta:

```

public void EscribirTextoOculta(string texto,
    short x, short y, byte r, byte g, byte b, Fuente fuente)
{
    Sdl.SDL_Color color = new Sdl.SDL_Color(r, g, b);
    IntPtr textoComoImagen = SdlTtf.TTF_RenderText_Solid(
        fuente.GetPuntero(), texto, color);
    if (textoComoImagen == IntPtr.Zero)
        Environment.Exit(5);

    Sdl.SDL_Rect origen = new Sdl.SDL_Rect(0, 0,
        anchoPantalla, altoPantalla);
    Sdl.SDL_Rect dest = new Sdl.SDL_Rect(x, y,
        anchoPantalla, altoPantalla);

    Sdl.SDL_BlitSurface(textoComoImagen, ref origen,
        pantallaOculta, ref dest);
}

```

Y ya lo podemos usar desde nuestro juego. Por una parte, cargaremos el tipo de letra (con el tamaño que deseemos) dentro del "inicializar":

```

void inicializar()
{
    h = new Hardware(800, 600, 24, pantallaCompleta);
    img = new Imagen("personaje.bmp", 50, 50);
    letra = new Fuente("FreeSansBold.ttf", 18);
    x = 400;
    y = 300;
}

```

Y escribiremos el texto que nos interese, en las coordenadas que queramos y con el color que nos apetezca, desde el método "dibujarElementos":

```

void dibujarElementos()
{
    h.BorrarPantallaOculta();
}

```

```

h.DibujarImagenOculta(img);
h.EscribirTextoOculta("Texto de ejemplo",
    200, 100, // Coordenadas
    255, 0, 0, // Color: rojo
    letra);
h.VisualizarOculta();
}

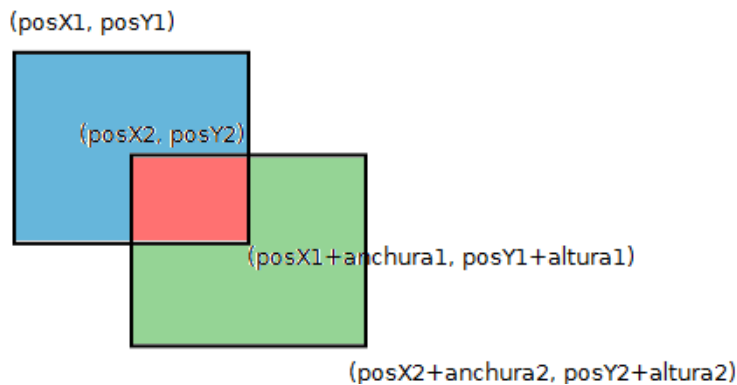
```

Ejercicios propuestos

(Ap4.6.1) Amplía el ejercicio Ap4.5.1, para que muestre una "puntuación" en pantalla (que, por ahora, siempre será cero).

Ap4.7. Colisiones simples

En cualquier juego habrá que comprobar colisiones: recoger premios, perder una vida si nos toca el enemigo o un disparo, etc. No basta con comparar la X y la Y de cada elemento, porque pueden solaparse aunque no estén exactamente en la misma posición.



Deberemos comprobar si el rectángulo que ocupa una imagen se superpone con el que ocupa la otra: si el rectángulo 2 (verde) está más a la derecha que el rectángulo 1 (azul), entonces su posición X (posX2) debe estar entre posX1 y posX1+anchura1. Si está más a la izquierda, estará entre posX1-anchura2 y posX1. Lo mismo ocurrirá con las coordenadas Y. Entonces, podemos crear un método en la clase Imagen que compruebe colisiones con otra imagen, así:

```

public bool ColisionCon(Imagen i)
{
    if ((x + ancho > i.x)
        && (x < i.x + i.ancho)
        && (y + alto > i.y)
        && (y < i.y + i.alto))
        return true;
    else
        return false;
}

```

Ahora podemos comprobar si dos imágenes se solapan. Por ejemplo, podemos ver si un enemigo toca a nuestro personaje, para que en ese momento acabe la partida:

```
if (imgPersonaje.CollisionCon(imgEnemigo))
{
    terminado = true;
}
```

O bien, que al tocar un premio obtengamos puntos y éste se recoloca en una posición distinta de la pantalla:

```
if (imgPersonaje.CollisionCon(imgPremio))
{
    puntos += 10;
    xPremio = (short)generadorAzar.Next(750);
    yPremio = (short)generadorAzar.Next(550);
    imgPremio.MoverA(xPremio, yPremio);
}
```

Para que el enemigo se mueva "por sí mismo", podemos indicar, además de su posición, su velocidad, de modo que el método "MoverElementos" vaya alterando su posición. Y podemos hacer que "rebote" en los extremos de la pantalla, cambiando el signo de su velocidad (de positiva a negativa, o viceversa) cuando los alcance:

```
xEnemigo += velocXEnemigo;
yEnemigo += velocYEnemigo;
imgEnemigo.MoverA(xEnemigo, yEnemigo);

if ((xEnemigo < 10) || (xEnemigo > 750))
    velocXEnemigo = (short) (-velocXEnemigo);

if ((yEnemigo < 10) || (yEnemigo > 550))
    velocYEnemigo = (short)(-velocYEnemigo);
```

El fuente completo podría ser así:

```
// sdl06.cs
// Sexto acercamiento a SDL: colisiones
// Introducción a C#, por Nacho Cabanes

using Tao.Sdl;
using System; // Para IntPtr (puntero: imágenes, etc)

public class Juego
{
    bool pantallaCompleta = false;
    Hardware h;
    Imagen imgPersonaje, imgEnemigo, imgPremio;
```

```

Fuente letra;
bool terminado = false;
short x, y;
short xEnemigo, yEnemigo, velocXEnemigo, velocYEnemigo;
short xPremio, yPremio;
int puntos;
Random generadorAzar;

void inicializar()
{
    h = new Hardware(800, 600, 24, pantallaCompleta);
    imgPersonaje = new Imagen("personaje.bmp", 32, 30);
    imgEnemigo = new Imagen("enemigo.bmp", 36, 42);
    imgPremio = new Imagen("premio.bmp", 36, 14);
    letra = new Fuente("FreeSansBold.ttf", 18);
    generadorAzar = new Random();
    x = 400; y = 300;
    xEnemigo = 100; yEnemigo = 100;
    velocXEnemigo = 3; velocYEnemigo = -3;
    xPremio = (short) generadorAzar.Next(750);
    yPremio = (short) generadorAzar.Next(550);
    imgPremio.MoverA(xPremio, yPremio);
    puntos = 0;
}

void comprobarTeclas()
{
    if (h.TeclaPulsada(Hardware.TECLA_ARR))
        y -= 4;
    if (h.TeclaPulsada(Hardware.TECLA_ABA))
        y += 4;
    if (h.TeclaPulsada(Hardware.TECLA_IZQ))
        x -= 4;
    if (h.TeclaPulsada(Hardware.TECLA_DER))
        x += 4;
    if (h.TeclaPulsada(Hardware.TECLA_ESC))
        terminado = true;
    imgPersonaje.MoverA(x, y);
}

void comprobarColisiones()
{
    // Si toca el premio: puntos y nuevo premio
    if (imgPersonaje.CollisionCon(imgPremio))
    {
        puntos += 10;
        xPremio = (short)generadorAzar.Next(750);
        yPremio = (short)generadorAzar.Next(550);
        imgPremio.MoverA(xPremio, yPremio);
    }

    // Si toca el enemigo: acabó la partida
    if (imgPersonaje.CollisionCon(imgEnemigo))
    {
        terminado = true;
    }
}

```

```

void moverElementos()
{
    xEnemigo += velocXEnemigo;
    yEnemigo += velocYEnemigo;
    imgEnemigo.MoverA(xEnemigo, yEnemigo);
    // Para que rebote en los extremos de la pantalla
    if ((xEnemigo < 10) || (xEnemigo > 750))
        velocXEnemigo = (short) (-velocXEnemigo);
    if ((yEnemigo < 10) || (yEnemigo > 550))
        velocYEnemigo = (short) (-velocYEnemigo);
}

void dibujarElementos()
{
    h.BorrarPantallaOculta();
    h.DibujarImagenOculta(imgPersonaje);
    h.DibujarImagenOculta(imgPremio);
    h.DibujarImagenOculta(imgEnemigo);
    h.EscribirTextoOculta("Puntos: "+puntos,
        200, 100, // Coordenadas
        255, 0, 0, // Color: rojo
        letra);
    h.VisualizarOculta();
}

void pausaFotograma()
{
    // Esperamos 20 ms
    h.Pausa(20);
}

bool partidaTerminada()
{
    return terminado;
}

public 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());
}
}

```


Puedes descargar todo el proyecto, incluyendo fuentes, imágenes y ficheros DLL desde

<http://www.nachocabanes.com/fich/descargar.php?nombre=SdlClases04.zip>

Ejercicios propuestos

(Ap4.7.1) Amplía el esqueleto Ap4.6.1, para que muestre varios premios en posiciones al azar. Cada vez que se toque un premio, éste desaparecerá y se obtendrán 10 puntos. Al recoger todos los premios, acabará la partida.

(Ap4.7.2) Amplía el ejercicio Ap4.7.1, para que haya un "margen de la pantalla" y una serie de obstáculos, que "nos maten": si el personaje los toca, acabará la partida.

(Ap4.7.3) Amplía el ejercicio Ap4.7.2, para que el movimiento sea continuo: el jugador no se moverá sólo cuando se pulse una flecha, sino que continuará moviéndose hasta que se pulse otra flecha (y entonces cambiará de dirección) o choque con un obstáculo (y entonces acabará la partida).

(Ap4.7.4) Amplía el ejercicio Ap4.7.3, para que el jugador sea una serpiente, formada por varios segmentos; cada vez que se recoga un premio, la serpiente se hará más larga en un segmento. Si la "cabeza" de la serpiente toca la cola, la partida terminará.

(Ap4.7.5) Mejora la jugabilidad del ejercicio Ap4.7.4: el jugador tendrá 3 vidas, en vez de una; si se recogen todos los premios, comenzará un nuevo nivel, con mayor cantidad de premios y de obstáculos.

(Ap4.7.6) Añade dificultad creciente al ejercicio Ap4.7.5: cada cierto tiempo aparecerá un nuevo obstáculo, y en cada nuevo nivel, el movimiento será un poco más rápido (puedes conseguirlo reduciendo la pausa de final del bucle de juego).

(Ap4.7.7) Añade al ejercicio Ap4.7.6 un tabla de mejores puntuaciones, que se guardará en fichero y se leerá al principio de cada nueva partida.

Ap4.8. 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 y más optimizados, 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` (y este cambio sólo afectaría al constructor de la clase `Imagen`):

```
imagen = SdlImage.IMG_Load("personaje.png");
```

Ejercicios propuestos

(Ap4.8.1) Mejora el juego de la serpiente, para que tenga una imagen JPG de fondo de la pantalla, y que los premios sean imágenes PNG con transparencia, a través de las que se pueda ver dicho fondo.

Ap4.9. ¿Por dónde seguir?

Este texto no pretende ser un curso de programación de videojuegos, de modo que muchas cosas quedan en el tintero. Aun así, vamos a mencionar algunas, para que el lector pueda intentar implementarlas o, al menos, saber por dónde podría profundizar:

- SDL, al igual que la mayoría de bibliotecas de juegos, tiene soporte para el uso del ratón y de joystick/gamepad. Para ver el estado del ratón se usaría `Sdl.SDL_GetMouseState` y para el joystick (o gamepad, o cualquier otro dispositivos similar, como un volante) se usaría `Sdl.SDL_JoystickGetAxis` para ver si se ha inclinado en alguno de los ejes y `Sdl.SDL_JoystickGetButton` para comprobar los botones.
- También permite reproducción de sonidos. La biblioteca básica soporta el formato WAV, sin comprimir, pero si preferimos formatos más compactos como MP3 o MID, podemos añadir a nuestra carpeta el fichero `SDL_mixer.dll`, que nos permitirá usar `SdlMixer.Mix_PlayMusic` para reproducir un sonido una vez, o varias, o de forma ininterrumpida.
- Si queremos dibujar líneas, puntos, rectángulos, círculos o elipses, podemos incluir también `Sdl_Gfx.dll`, con funciones como `SdlGfx.rectangleRGBA`.
- Si queremos que una imagen sea "animada", es decir, formada por varios fotogramas, podemos usar un array de imágenes. Para simplificar su manejo, podemos crear una clase `ElementoGráfico`, de modo que sea una extensión del concepto de imagen: un elemento gráfico podrá contener una secuencia de imágenes, o incluso varias secuencias distintas, según la dirección en que se esté moviendo el personaje. Entre sus métodos, podría existir un `"SiguienteFotograma"`, que pase a mostrar la siguiente imagen de la animación.

- Los fondos repetitivos que aparecen en la mayoría de juegos de plataformas clásicos se pueden realizar mediante un array de imágenes. Para saber si es posible moverse a una cierta posición de la pantalla o no, comprobaríamos antes si supondría colisionar con alguna de esas imágenes.
- Algunos movimientos repetitivos del personaje (caer, saltar) pueden suponer que se siga moviendo aunque el usuario no pulse ninguna tecla. Ese caso se trataría dentro del método "MoverElementos" y sería básicamente una comprobación de una o dos condiciones que dan lugar al siguiente paso: `if (cayendo && posibleBajar) y+=2;`
- A medida que el juego crece, puede ser interesante descomponerlo como una serie de clases que se interrelacionan: juego, personaje, enemigo, disparo, pantallaDeFondo, pantallaDeAyuda. Así, cada clase tiene unas responsabilidades más delimitadas, y el programa resultante es más fácil de mantener.

Ejercicios propuestos

(Ap4.9.1) Crea una versión del juego de la serpiente en la que la lógica esté repartida entre varias clases que colaborar: JuegoSerpiente (que contendrá el Main), Presentacion, PantallaAyuda, PantallaCreditos, Nivel, Personaje, Premio.

Apéndice 5. Contacto con los entornos gráficos

Ap5.1. Creación de formularios, botones y etiquetas

En C# podemos crear con una cierta facilidad programas en entornos gráficos, con menús botones, listas desplegables, etc.

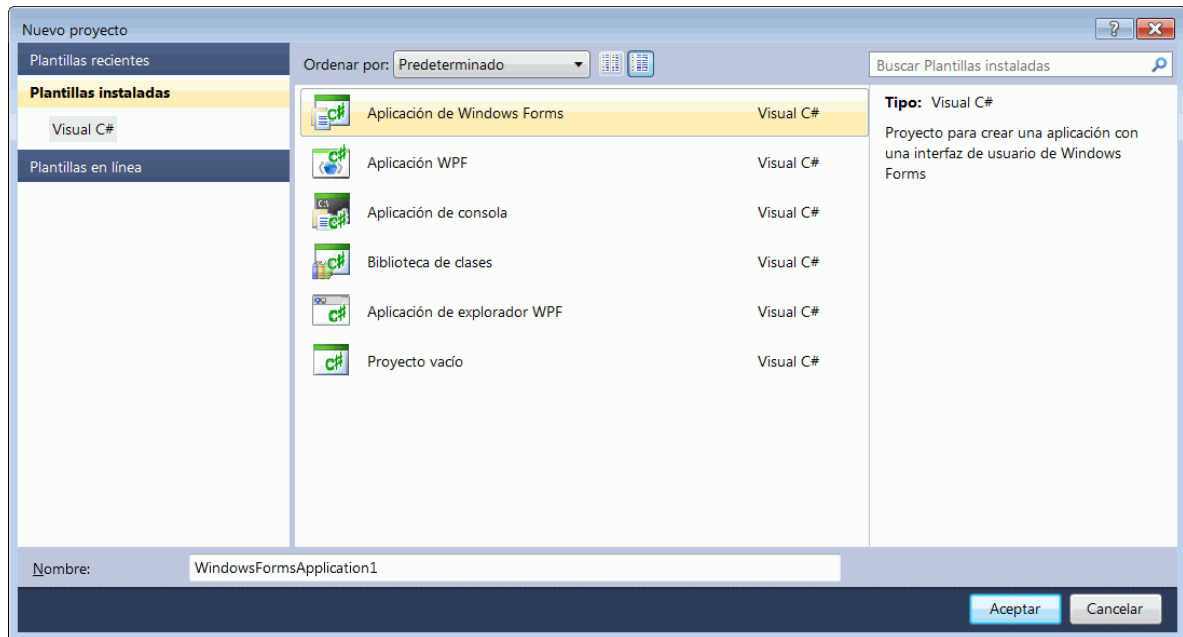
La forma más cómoda de conseguirlo es usando herramientas que incluyan un editor visual, como Visual Studio o SharpDevelop.

SharpDevelop necesita un ordenador menos potente que Visual Studio y tiene un manejo muy similar a éste. Aun así, dado que la versión Express de Visual Studio es gratis para uso personal y se mueve con una soltura razonable en un ordenador "moderno" (más de 1 Gb de memoria RAM, procesador de un núcleo de 2 Ghz o más, o bien procesador de doble núcleo), vamos a centrarnos en su manejo.

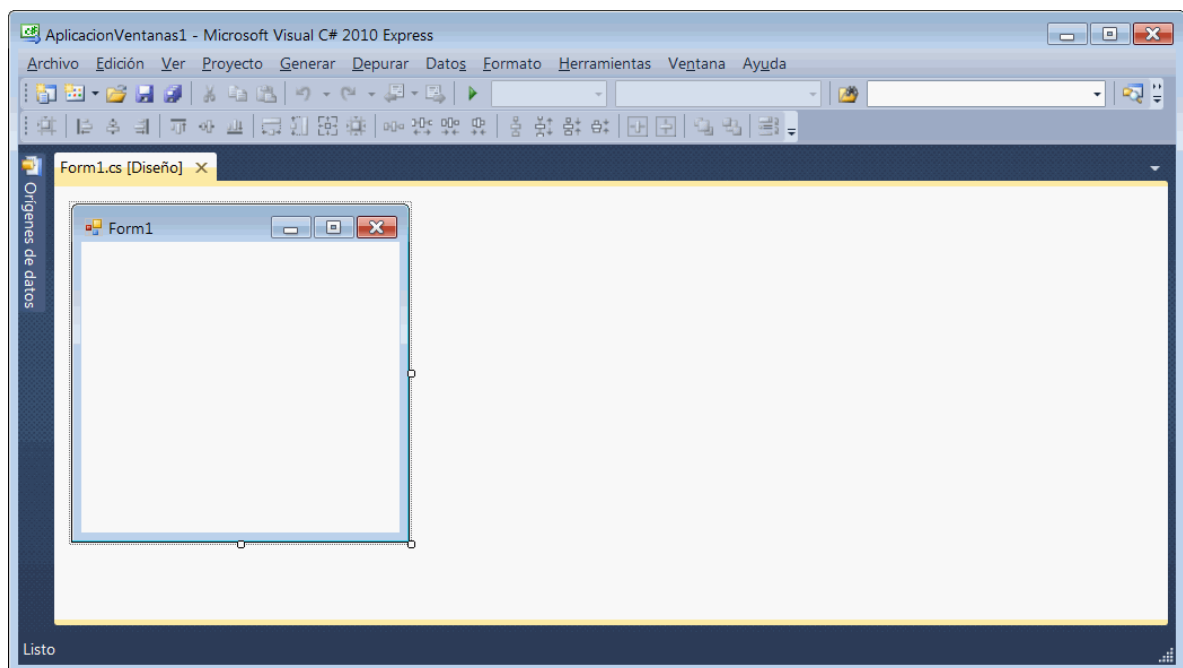
Cuando lanzamos Visual Studio 2010, aparece una pantalla que nos muestra los últimos proyectos que hemos realizado y nos da la posibilidad de crear un nuevo proyecto:



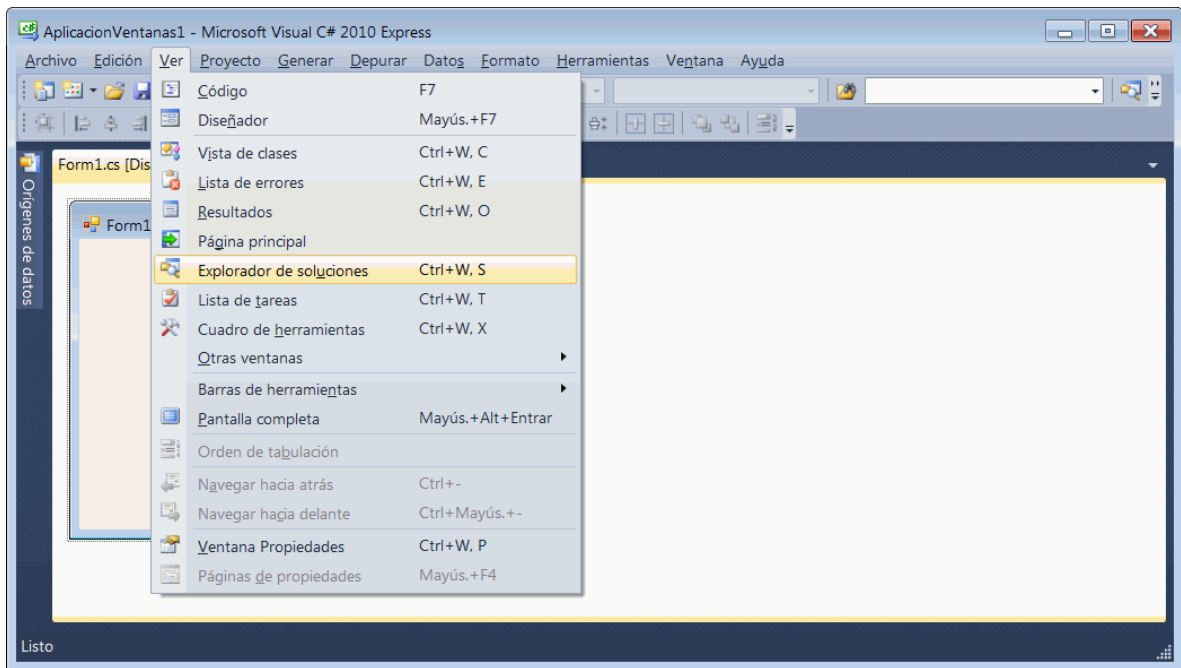
Al elegir un "Nuevo proyecto", se nos preguntará de qué tipo queremos que sea, así como su nombre. En nuestro caso, será una **"Aplicación de Windows Forms"**, y como nombre se nos propondrá algo como "WindowsFormsApplication1", que nosotros podríamos cambiar por "AplicacionVentanas1" (como siempre, sin espacios ni acentos, que suelen ser caracteres no válidos en un identificador - aunque Visual Studio sí permitiría los acentos-):



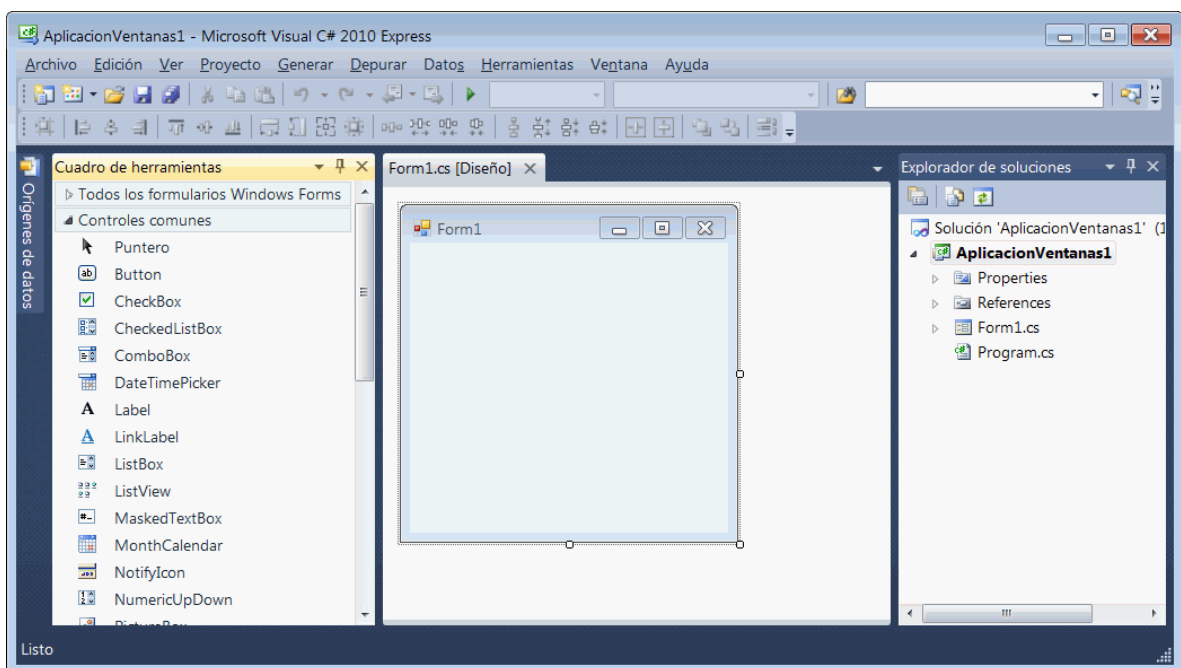
Nos aparecerá un formulario ("una ventana") vacío:



Si nuestra pantalla está así de vacía, nos interesará tener un par de herramientas auxiliares. La primera es el "Explorador de soluciones", que nos permitirá pasear por las distintas clases que forman nuestro proyecto. La segunda es el "Cuadro de herramientas", que nos permitirá añadir componentes visuales, como botones, menús y listas desplegables. Ambos los podemos añadir desde el menú "Ver":



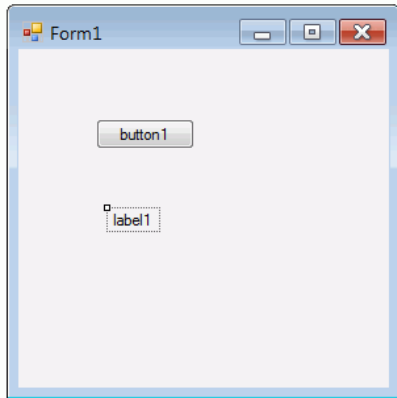
De modo que ahora nuestra pantalla debería ser algo así:



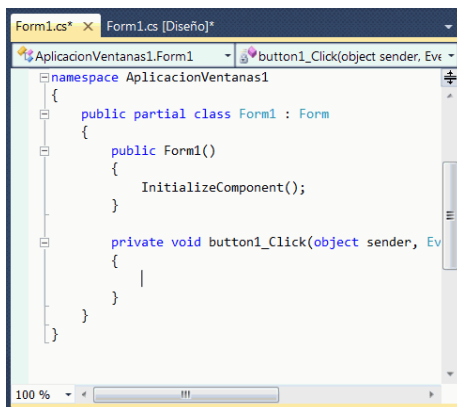
Como primer ejemplo, vamos a añadir a nuestro formulario un botón (Button) y una etiqueta de texto (Label), para hacer un primer programa que cambie el texto de la etiqueta cuando pulsemos el botón.

Hacemos clic en el componente Button del Cuadro de Herramientas (panel izquierdo) y luego clic dentro del formulario, para que aparezca un botón en esa

posición. De igual modo, hacemos clic en Label para indicar que queremos una etiqueta de texto, y luego clic en cualquier punto de nuestro formulario. Nuestra ventana debería estar quedando así:



Ahora vamos a hacer que nuestro programa responda a un clic del ratón sobre el botón. Para eso, hacemos doble clic sobre dicho botón, y aparecerá un esqueleto de programa que podemos rellenar:



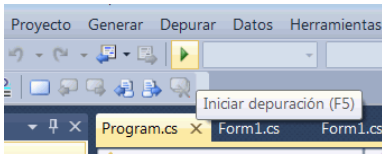
En ese hueco indicaremos lo que debe ocurrir cuando se pulse el botón ("button1_Click"). En nuestro caso, simplemente será cambiar el texto del "label", así:

```

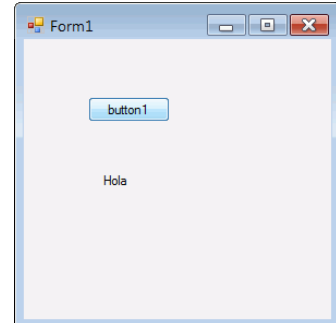
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Hola";
}

```

Antes de ver con más detalle por qué ocurre todo esto y mejorarlo, vamos a comprobar que funciona. Pulsamos el botón de "Iniciar depuración" en la barra de herramientas:



Y nos aparecerá nuestra ventana, lista para probar. Si pulsamos el botón, cambiará el texto de la etiqueta:



Antes de seguir, deberíamos grabar los cambios, porque algunas versiones de Visual Studio permiten "compilar a memoria", de modo que nuestro programa quizá todavía no está guardado.

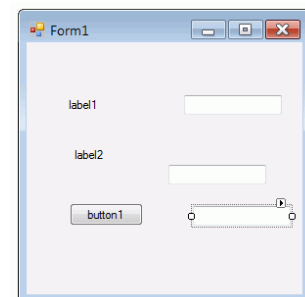
Ahora vamos a crear un segundo proyecto, un poco más elaborado y que nos ayude a entender mejor el funcionamiento del entorno.

Ap5.2. Cambios de apariencia. Casillas de texto para sumar dos números

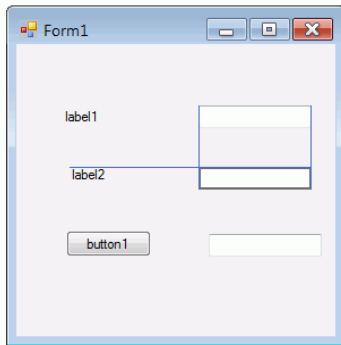
Hemos visto cómo colocar un botón y una casilla de texto, pero no hemos cambiado el texto del botón, ni su tamaño, ni el tipo de letra. Ahora vamos a hacer un programa que calcule y muestre la suma de dos números, y cuidaremos un poco más la apariencia.

Comenzamos por crear un nuevo proyecto llamado (por ejemplo) "AplicacionVentanas2", que también será una "Aplicación de Windows Forms".

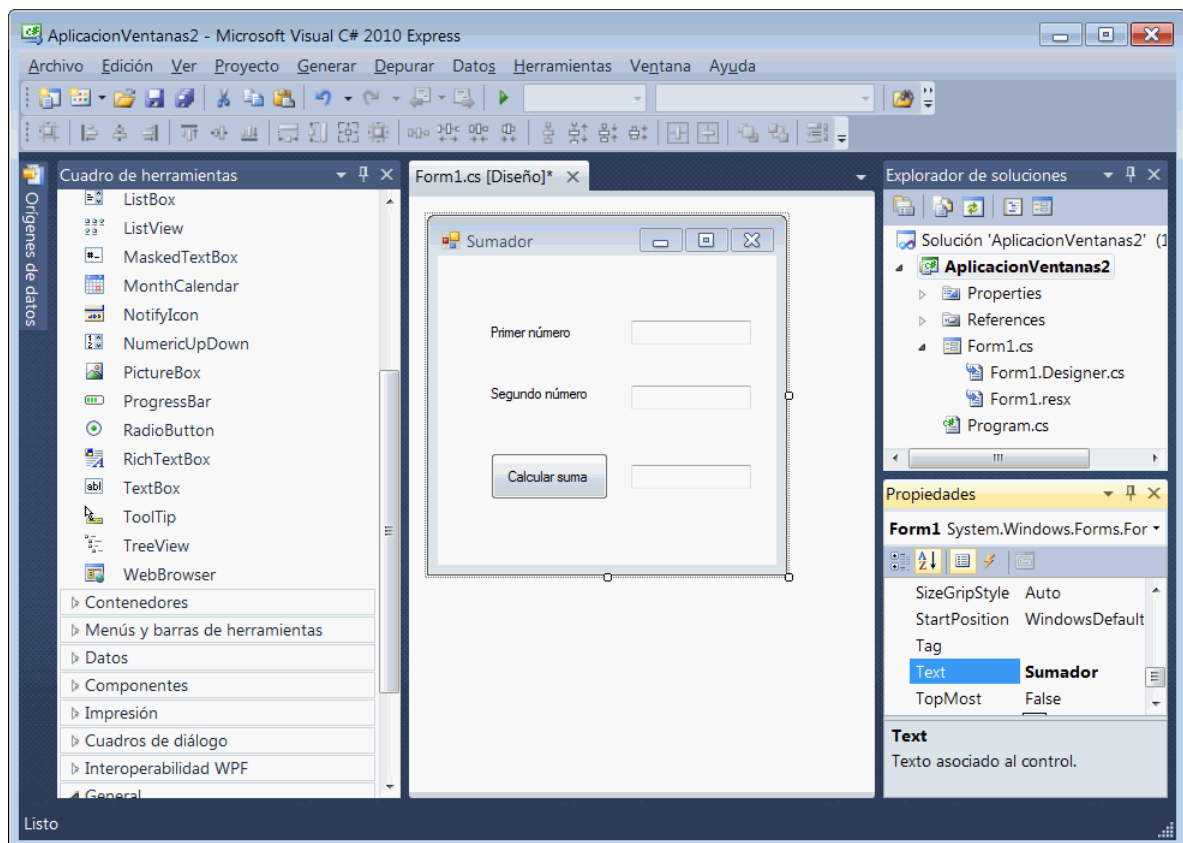
En el formulario, añadiremos dos Label para incluir textos explicativos, tres TextBox (casillas de introducción de texto) para los datos y el resultado, y también un botón para que se calcule dicho resultado. El resultado todavía será feo y, posiblemente, con los componentes desalineados:



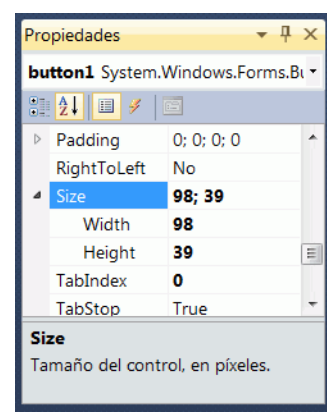
Vamos a comenzar por alinear los componentes. Si pinchamos y arrastramos uno de ellos con el ratón, cuando lo movamos es fácil que aparezcan unas rayas para avisarnos de que en ese momento están alineados varios componentes:



Una vez que estén colocados, vamos a cambiar sus propiedades. Sabemos cambiar el texto de un Label desde código, pero también podemos hacerlo desde el diseñador visual. Basta con tener la ventana de "Propiedades", que podemos obtener al pulsar el botón derecho sobre uno de los componentes o desde el menú "Ver". Ahí tenemos la propiedad "Text", que es el texto que cada componente muestra en pantalla. Haciendo clic en cada uno de ellos, podemos cambiar el texto de todos (incluyendo el "nombre de la ventana"):



La anchura de las casillas de texto y el tamaño del botón los podemos cambiar simplemente pinchando y arrastrando con el ratón, pero también tenemos una propiedad "Size", con componentes X e Y:



Algunas de las 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 el componente.
- BackColor, el color de fondo.
- 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).
- Font, el tipo de letra.
- ReadOnly, para poder hacer que un TextBox sea sólo de lectura y no se pueda modificar su contenido.

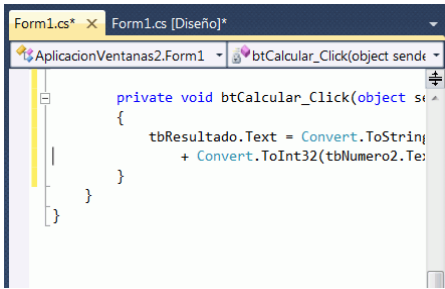
En nuestro caso, vamos a hacer los siguientes cambios:

- Las etiquetas de texto, en vez de label1 y label2, se llamarán lbNumero1 y lbNumero2.
- De igual modo, las casilla de texto, en vez de textBox1, textBox2, y textBox3 se llamarán tbNumero1, tbNumero2 y tbResultado.
- El botón, en vez de Button1, pasará a llamarse btCalcular.
- El formulario se llamará frmSumador.
- La casilla del resultado usará el mismo tipo de letra, pero en negrita (Bold). Además, será sólo de lectura.

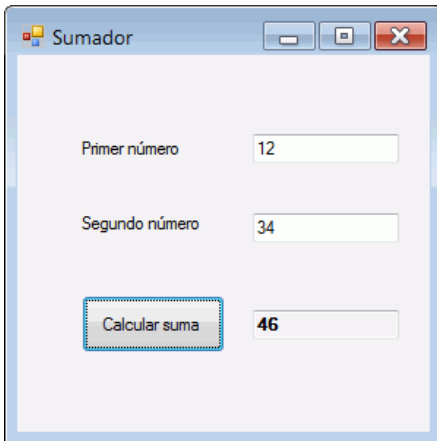
Nuevamente, para hacer que se calcule el resultado al pulsar el botón, hacemos doble clic sobre él. En este caso, el texto de tbResultado será la suma de los de los otros TextBox. Eso sí, como las casillas de texto contienen cadenas, habrá que convertir a int32 para poder sumar, y luego volver a convertir a string para almacenar el resultado:

```
private void btCalcular_Click(object sender, EventArgs e)
{
    tbResultado.Text = Convert.ToString( Convert.ToInt32(tbNumero1.Text)
        + Convert.ToInt32(tbNumero2.Text) );
}
```

Podemos volver a la vista de diseño en cualquier momento, haciendo clic en la pestaña que hay en la parte superior:



La apariencia del programa, ya en funcionamiento, sería algo como:



Es muy mejorable, pero algunas de las mejoras son parte de los ejercicios propuestos.

Ejercicios propuestos:

(Ap5.2.1) Mejora el ejemplo anterior para que las casillas tengan el texto alineado a la derecha y para que no falle si las casillas están vacías o contienen texto en lugar de números.

(Ap5.2.2) Crea una nueva versión del programa, que no solo muestre la suma, sino también la resta, la multiplicación, la división y el resto de la división.

(Ap5.2.3) Un programa que muestre una ventana con un recuadro de texto, un botón y 3 etiquetas. En el recuadro de texto se escribirá un número (en sistema decimal). Cada vez que se pulse el botón, se mostrará en las 3 etiquetas de texto el equivalente de ese número en binario, octal y hexadecimal.

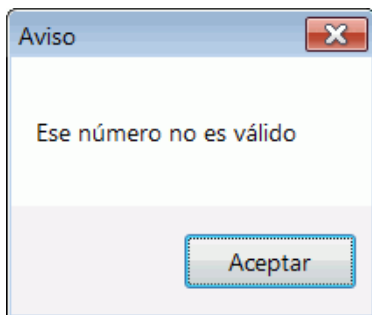
(Ap5.2.4) Haz una variante del programa anterior, que no use un botón para convertir el número a binario, octal y hexadecimal. En vez de eso, cada vez que en el recuadro de texto se pulse una tecla, se mostrará automáticamente en las 3 etiquetas de texto el equivalente del número actual. Pista: al hacer doble clic sobre una casilla de texto, aparecerá el evento "TextChanged", que es el que se lanza cuando se modifica el texto que contiene un TextBox.

Ap5.3. 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.

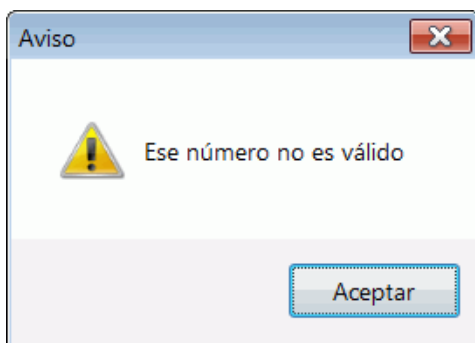
Una forma alternativa y sencilla de conseguirlo es usando "ventanas de mensaje". Éstas se pueden crear llamando a "MessageBox.Show", que tiene varias sintaxis posibles, según el número de parámetros que queramos utilizar. Por ejemplo, podemos mostrar un cierto texto de aviso en una ventana que tenga un título dado:

```
MessageBox.Show("Ese número no es válido", "Aviso");
```



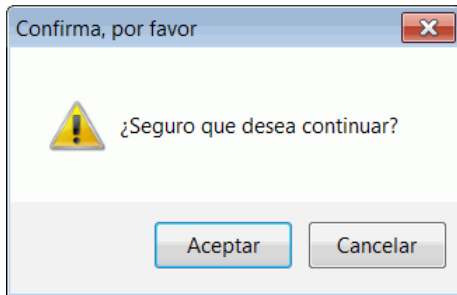
La segunda variante es indicar además qué botones queremos mostrar, y qué iconos de aviso:

```
MessageBox.Show("Ese número no es válido", "Aviso",  
    MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
```



Y la tercera variante permite indicar además el que 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 por defecto (MessageBoxDefaultButton) 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.

Ejercicios propuestos:

(Ap5.3.1) Mejora el ejercicio Ap5.2.1, para que muestre un mensaje de aviso si las casillas están vacías o contienen texto en lugar de números.

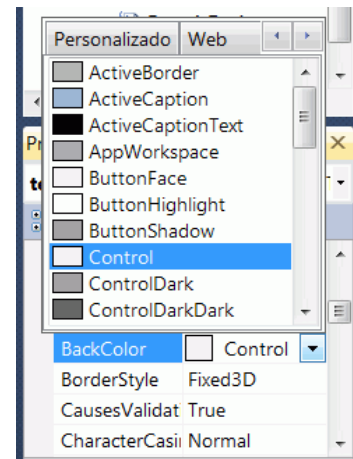
Ap5.4. Una aplicación con dos ventanas

Si queremos una ventana auxiliar, que permita al usuario introducir varios datos, deberemos crear un segundo formulario, y llamarlo desde la ventana principal. Vamos a ver los pasos necesarios.

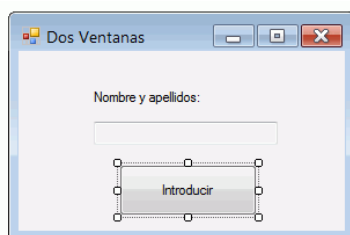
En primer lugar, crearemos un nuevo proyecto, de tipo "Aplicación de Windows Forms", que llamaremos (por ejemplo) "DosVentanas" y entraremos a la vista de diseño para crear la que será la ventana principal de nuestra aplicación.

Vamos a crear en ella 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.

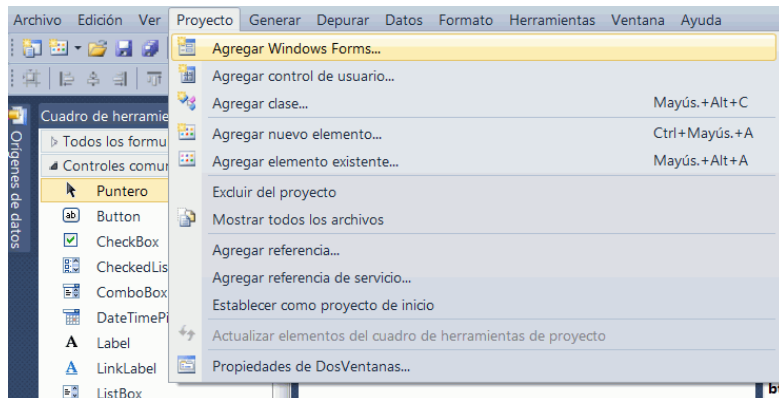
Comenzamos por crear el "TextBox" que mostrará el texto, y el "Label" que aclarará qué es ese texto. Cambiamos el "Text" de la etiqueta para que muestre "Nombre y apellidos", y cambiamos la propiedad "ReadOnly" de la casilla de texto para que sea "true" (verdadero), de modo que no se pueda escribir en esa casilla. También podemos cambiar el color de la casilla, para que sea más evidente que "no es una casilla normal". Visual Studio 2010 lo hace automáticamente, pero si un entorno más antiguo no lo hace por nosotros, podríamos pedir que fuera 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".



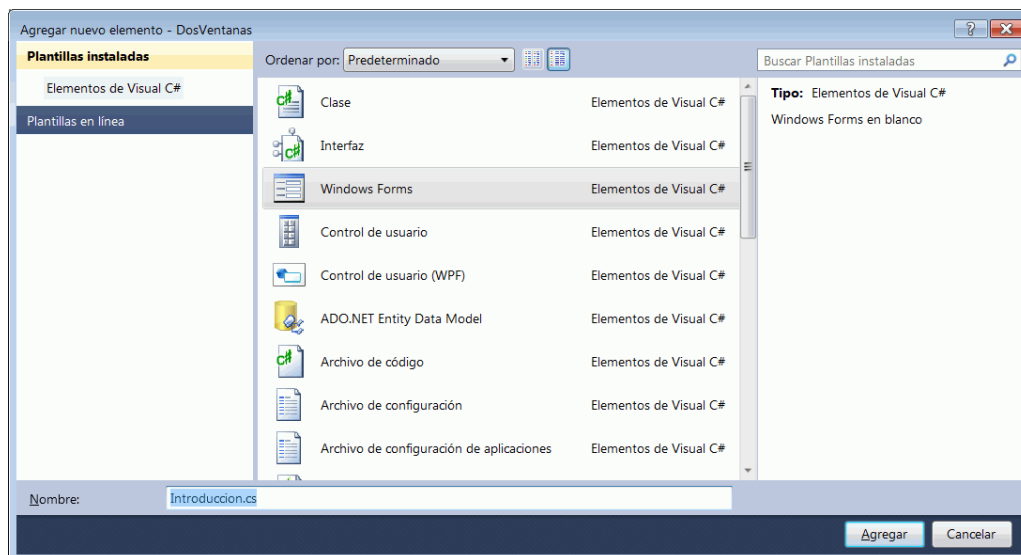
La apariencia de nuestra ventana debería ser parecida a ésta:



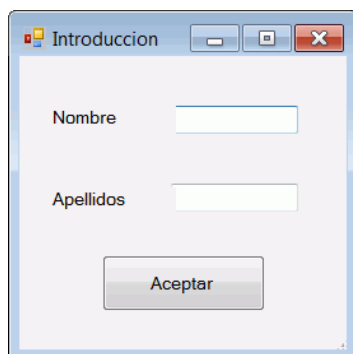
Para crear el segundo formulario (la ventana auxiliar), usamos la opción "Agregar Windows Forms", del menú "Proyecto":



y le damos un nombre, por ejemplo "Introduccion.cs":



Nos aparecerá una nueva ventana, en su vista de diseño. 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 GetNombreApellido()
{
    return tbNombre.Text + " " + tbApellidos.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:

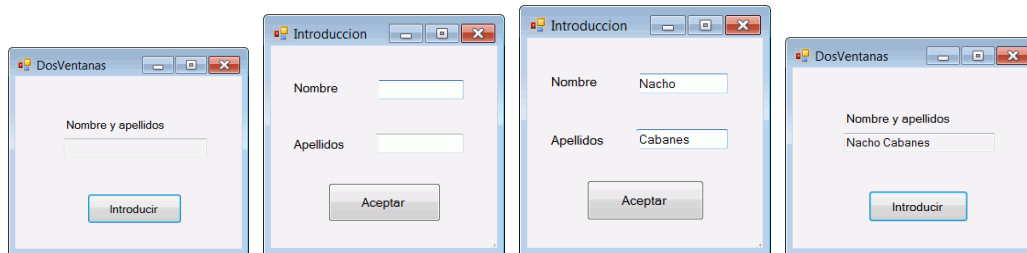
```
public MainForm()
{
    InitializeComponent();
    ventanaIntro = new Introduccion();
}
```

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();
    tbNombreApellido.Text = ventanaIntro.GetNombreApellido();
}
```


}

El resultado sería una secuencia como esta:



Ejercicios propuestos

(Ap5.4.1) Crea un programa que descomponga un número como producto de sus factores primos, usando ventanas.

(Ap5.4.2) Crea un programa que muestre una casilla de texto en la que el usuario puede introducir frases. Todas estas frases se irán guardando en un fichero de texto.

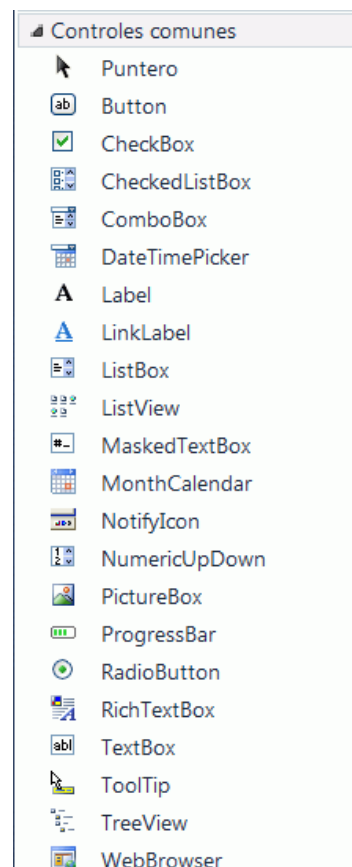
(Ap5.4.3) Crea una versión "con ventanas" de la "base de datos de ficheros" (ejemplo 04_06a).

Ap5.5. Otros componentes visuales

Por supuesto, hay muchos más controles visuales que los que hemos visto, pero en este texto no se pretende dominar todos los componentes existentes, sino dar una alternativa a la entrada y salida convencional a través de consola.

Entre los otros muchos componentes existentes, y que quedan a la curiosidad del lector, cabe mencionar, por ser frecuentes y razonablemente sencillos de utilizar:

- CheckBox, para permitir al usuario escoger entre varias opciones, y RadioButton, cuando sólo se puede elegir una de dichas opciones.
- ListBox, para mostrar una lista de valores, y ComboBox para permitir introducir un valor o escoger de una lista desplegable.
- PictureBox, para mostrar una imagen.
- ProgressBar, para una barra de progreso.



- TextBox tiene una propiedad "MultiLine", para indicar si es de múltiples líneas. En ese caso, su contenido no se obtendría con "Text", sino recorriendo sus "Lines".
- También, dentro de otras categorías, podemos crear paneles para agrupar elementos, incluir menús y barras de herramientas, mostrar ventanas de diálogo estándar (como la de abrir un fichero, la de guardarlo, la de navegar por carpetas, la de elegir un color, la de elegir un tipo de letra), así como realizar otras muchas tareas de mayor complejidad, como crear componentes semiautomáticos para acceder a bases de datos, mostrar vistas previas de impresión, crear temporizadores, monitorizar cambios en el sistema de ficheros, etc.

Ap5.6. 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. 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 icono (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 rellenar el interior de una elipse.
- FillPolygon, para rellenar el interior de un polígono.
- FillRectangle, para rellenar el interior de un rectángulo.

Por otra parte, 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 quizá no sea la misma que el fuente), y puede estar en formato BMP, GIF, PNG, JPG o TIFF.

Se puede encontrar más detalles sobre la clase Graphics en la referencia en línea (MSDN), por ejemplo en la página

http://msdn.microsoft.com/es-es/library/system.drawing.graphics_methods.aspx

Ejercicios propuestos

(Ap5.6.1) Crea una versión del "juego del ahorcado" (ejercicio 5.9.1.2) basada en Windows Forms, en la que, a medida que el usuario falle, se vaya "dibujando un patíbulo".

Revisiones de este texto

- 0.99zz, de 22-ene-15. Nueva revisión, con la mayoría de temas ligeramente ampliados, con más ejemplos y con más ejercicios propuestos. Los apartados sobre SDL y Windows Forms pasan a ser apéndices.
- 0.99, de 30-dic-12. La mayoría de temas están ligeramente ampliados y con más ejercicios propuestos. Incluidos apartados sobre persistencia y sobre acceso a bases de datos con SQLite. Muchos más detalles sobre SDL y algunos más sobre Windows Forms.
- 0.98, de abr-11. Lanzamiento de excepciones. Resultado de un MessageBox. Errata en "operaciones de bits" (suma lógica).
- 0.97, de 01-feb-11. 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

-	@
-, 34	@, 109, 383
	[
--, 94	[.] (arrays), 123
	[] (arrays), 113
!	[Serializable], 275
!, 52	\
!=, 49	\, 109
	^
#	^, 356
#, 104	{
%	{ y }, 17, 48
%, 34	
%=, 96	, 356
	, 52
&	~
&, 356	~, 214, 356
& (dirección de una variable), 330	+
&&, 52	+, 34
	++, 68, 94
*	+=, 96
*, 34	
*=, 96	<
,	<, 49
,, 364	<<, 356
.	<> (Generics), 326
.Net, 13	=
/	=, 37
/, 34	-=, 96
/**, 382	==, 49
//, 42	
///, 382	>
/=, 96	>, 49
:	>>, 356
:(goto), 82	
?	0
?, 58	0x (prefijo), 106

A

Abs, 172
 Acceso aleatorio, 253
 Acceso secuencial, 253
 Acos, 172
 Add (ArrayList), 316
 AddDays, 337
 Al azar, números, 170
 Aleatorio, acceso, 253
 Aleatorios, números, 170
 algoritmo, 14
 alto nivel, 9
 and, 356
 Añadir a un array, 118
 Añadir a un fichero, 247
 Apilar, 312
 Aplicación Windows, 420
 Append, 271
 AppendText, 247
 Arco coseno, 172
 Arco seno, 172
 Arco tangente, 172
 args, 178
 Argumentos de un programa, 177
 Aritmética de punteros, 334
 array, 113
 Array de objetos, 221
 ArrayList, 316
 Arrays bidimensionales, 122
 Arrays de struct, 126
 arreglo, 113
 ASCII, 387, 389, 400, 420
 Asignación de valores, 37
 Asignación en un "if", 54
 asignaciones múltiples, 96
 Asin, 172
 Atan, 172
 Atan2, 172
 azar, 170

B

BackgroundColor, 340
 bajo nivel, 10
 base, 230
 Base numérica, 105
 Bases de datos, 292
 Bases de datos con SQLite, 289
 BaseStream, 262
 BASIC, 9
 Begin (SeekOrigin), 259
 Binario, 105, 391
 BinaryReader, 261
 BinarySearch, 318
 BinaryWriter, 269
 bit, 386
 Bloque de programa, 70
 BMP (tipo de fichero), 263
 bool, 111
 Booleanos, 111, 121
 Borrar en un array, 118
 borrar la pantalla, 340
 break, 60
 bucle de juego, 409
 bucle sin fin, 69
 Bucles, 63

Bucles anidados, 69
 bug, 369
burbuja, 145
 Buscar en un array, 118
 Button, 422
 byte, 93, 385

C

C, 10
 C#, 10
 Cadena modificable, 138
 Cadenas de caracteres, 128
 Cadenas de texto, 110
 Cambio de base, 105
 Campo (bases de datos), 290
 Carácter, 107
 Carpetas, 248
 case, 60
 Caso contrario, 50
 catch, 88, 250

Ch

char, 60, 107

C

cifrar mensajes, 358
 Cifras decimales, 103
 Cifras significativas, 99
 class, 18
 Clear, 340
 Close, 243
 código máquina, 9
 Códigos de formato, 103
 Cola, 314
 Colisiones entre imágenes, 413
 Color de texto, 340
 Coma (operador, for), 364
 Coma fija, 97
 Coma flotante, 97
 Comentarios
 recomendaciones, 378
 Comentarios de documentación, 381
 Comillas (escribir), 109
 Comparación de cadenas, 137
 CompareTo, 137
 compiladores, 12
 Compilar con mono, 31
 Complemento, 356
 Complemento a 1, 398
 Complemento a 2, 398
 Consola, 340
 Console, 17, 340
 ConsoleKeyInfo, 341
 constantes, 358
 constructor, 212
 Contador, 65
 Contains, 132, 313, 320
 Contains (Hash), 321
 ContainsKey, 321
 ContainsValue, 320
 continue, 77
 Convert, 93, 100
 Convert.ToInt32, 43
 Convertir a binario, 105

Convertir a hexadecimal, 105
 Cos, 172
 Coseno, 172
 Coseno hiperbólico, 172
 Cosh, 172
 Create, 267
 CreateDirectory, 344
 CreateNew, 271
 CreateText, 242
 Current (SeekOrigin), 259

D

DataGridView, 310
 DateTime, 337
 Day, 337
 debug, 369
 decimal, 99
 Decimal (sistema de numeración), 391
 Declarar una variable, 36
 Decremento, 94
 default, 60
 Depuración, 369
 Dequeue, 314
 Desapilar, 312
 Desbordamiento, 36
 Descomposición modular, 156
 Desplazamiento a la derecha, 356
 Desplazamiento a la izquierda, 356
 destructor, 214
 Diagramas de Chapin, 83
 Diagramas de flujo, 55
 Dibujo con Windows Forms, 434
 Diccionario, 319
 Dinámica, memoria, 311
 Dirección de memoria, 329
 Directorios, 344
 DirectoryInfo, 344
 Diseño modular de programas, 156
 Distinto de, 49
 División, 34
 do ... while, 65
 Doble precisión, 97, 98
 Documentación, 377
 Documentación desde comentarios, 381
 Dos dimensiones (array), 122
 Dos ventanas (Windows Forms), 430
 Dot Net Framework, 13
 double, 99
 Doxygen, 381
 DrawLine, 435

E

E, 172
 ejecutable, 12
 elevado a, 172
 else, 50
 Encolar, 314
 End (SeekOrigin), 259
 Enqueue, 314
 ensamblador, 10
 ensambladores, 11
 enteros negativos, 397
 Entorno, 347, 348
 enum, 358
 Enumeraciones, 358

Enumeradores, 322
 Environment, 348
 Environment.Exit, 178
 ERRORLEVEL, 178
 Escritura indentada, 49
 Espacios de nombres, 354
 Estructuras, 125
 Estructuras alternativas, 47
 Estructuras anidadas, 127
 Estructuras dinámicas, 311
 Estructuras repetitivas, 63
 Euclides, 177
 Excepciones, 88, 250
 EXE, 12
 Exists (ficheros), 249
 Exit, 178
 Exp, 172
 Exponencial, 172
 Expresiones regulares, 361

F

factorial, 175
 false, 111, 121
 falso, 111, 121
 Fecha y hora, 337
 Fibonacci, 176
 Fichero físico, 253
 Fichero lógico, 253
 Ficheros, 242
 Ficheros binarios, 253
 Ficheros de texto, 242
 Ficheros en directorio, 345
 FIFO, 314
 File.Exists, 249
 FileAccess.ReadWrite, 271
 FileStream, 258
 fin de fichero, 245
 fixed, 335
 float, 99
 for, 67
 foreach, 139
 ForegroundColor, 340
 Formatear números, 102, 103
 fuente, 16
 Función de dispersión, 320
 Funciones, 156
 Funciones matemáticas, 171
 Funciones virtuales, 225

G

Generics, 326
 get, 360
 Get, 189
 GetEnumerator, 322
 GetFiles, 345
 GetKey, 319
 gigabyte, 385
 Global (variable), 163
 gmcs, 31
 goto, 82
 goto case, 60

H

Hash, 320

Hexadecimal, 105, 395
Hora, 337

I

Identificadores, 40
if, 47
Igual a, 49
Imágenes con SDL, 400
Incremento, 94
IndexOf, 132
Inseguro (bloque "unsafe"), 330
Inserción directa, 146
Insert, 133, 316
Insertar en un array, 118
int, 36, 37, 93
Internet, 349
intérprete, 12
Interrumpir un programa, 178
Introducción de datos, 43
IOException, 271
iterativa, 176

J

JavaDoc, 383
JPG, 417
Juegos, 400

K

Key, 341
KeyAvailable, 340
KeyChar, 341
kilobyte, 385

L

Label, 424
LastIndexOf, 132
Lectura y escritura en un mismo fichero, 271
Length (cadenas), 130
Length (fichero), 260
lenguaje C, 10
lenguaje máquina, 9
LIFO, 312
Línea de comandos, 177
Líneas, dibujar, 434
List, 326
lista, 316

LI

llaves, 17

L

Local (variable), 163
Log, 172
Log10, 172
Logaritmo, 172
long, 93
Longitud de una cadena, 130

M

Main, 17
máquina virtual, 12
matemáticas, funciones, 171
matriz, 113
Mayor que, 49
mayúsculas, 133
Mayúsculas y minúsculas, 40
mcs, 31
megabyte, 385
Memoria dinámica, 311
Menor que, 49
MessageBox, 428
métodos, 219
Mientras (condición repetitiva), 63
Modificar parámetros, 167
Módulo (resto de división), 34
Mono, 16, 19
MonoDevelop, 204
Month, 337
Mostrar el valor de una variable, 38
MoveNext, 323
Multiplicación, 34

N

n, 108
namespace, 354
Negación, 34
Negativos (números, representación), 397
Net, 13
NetworkStream, 350
new (objetos), 187
new (redefinir métodos), 206
Next (números al azar), 170
nibble, 392
No, 52
not, 356
Notepad++, 33
Now, 337
null (fin de fichero), 246
Números aleatorios, 170
números enteros, 34
Números reales, 97

O

0, 52
Objetos, 180
octal, 393
Octal, 105
Ocultación de datos, 189
OpenOrCreate, 271
OpenText, 244
OpenWrite, 267
Operaciones abreviadas, 96
Operaciones aritméticas, 34
Operaciones con bits, 356
operador coma, 364
Operador condicional, 58
Operadores, 34
Operadores lógicos, 52
Operadores relacionales, 49
Operator (sobrecarga de operadores), 237
or, 356
Ordenaciones simples, 145

Overflow, 36
override, 225

P

Palabra reservada, 40
Parámetros de Main, 177
Parámetros de una función, 158
parámetros por referencia, 168
parámetros por valor, 167
Pascal, 9
Paso a paso (depuración), 370
Pausa (Sleep), 338
Peek, 313
Persistencia de objetos, 274
Pi, 172
Pila, 312
PNG, 417
Polimorfismo, 215
POO, 180
pop, 312
Posición del cursor, 340
Posición en el fichero, 259
postdecremento, 95
postincremento, 95
Potencia, 172
Pow, 172
Precedencia de los operadores, 35
predecremento, 95
preincremento, 95
Prioridad de los operadores, 35
private, 209
Process.Start, 347
Producto lógico, 356
programa, 9
Programación orientada a objetos, 180
Propiedades, 360
protected, 208
Prueba de programas, 372
Pseudocódigo, 14
public, 18
public (struct), 125
Punteros, 311, 329
Punto Net, 13
Puntos de ruptura, 372
push, 312
Python, 10

Q

Queue, 314

R

Raíz cuadrada, 172
Random, 170
Rango de valores (enteros), 93
Read (FileStream), 258
Readkey, 340
ReadLine, 43, 44, 45, 46
ReadLine (fichero), 244
real (tipo de datos), 97
recolector de basura, 335
Recursividad, 174
Red local, 349
Redondear un número, 172
ref, 168

Referencia (paso de parámetros), 168
Regex, 362
Registro, 253
Registros, 125
Remove, 133
Repetir...mientras, 65
Replace, 133
Reserva de espacio, 333
Resta, 34
Resto de la división, 34
Retorno de Main, 177
return, 161
Round, 172

S

Sangrado, 49
sbyte, 93
SDL, 400
Secuencial, acceso, 253
Secuencias de escape, 108
Seek, 259
SeekOrigin, 259
Selección directa, 145
Seno, 172
Sentencias compuestas, 48
Serializable, 275
set, 360
Set, 189
SetByIndex, 320
SetCursorPosition, 340
SharpDevelop, 420
short, 93
Si no, 50
Signo y magnitud, 397
Simple precisión, 98
Sin, 172
Sistema (información), 348
Sistema (llamadas), 347
Sistema binario, 391
Sistema de numeración, 105
Sistema decimal, 391
Sleep, 338
SOAP, 283
Sobrecarga, 215
Sobrecarga de operadores, 237, 239
Sort, 316
SortedList, 319
Split, 135
SQL, 289
SQLite, 292
Sqrt, 172
Stack, 312
stackalloc, 333
static, 219
StreamReader, 244
StreamWriter, 242
string, 62, 110
String.Compare, 137
StringBuilder, 138
struct, 125
struct anidados, 127
Subcadena, 131
Subcadenas, 135
Substring, 131
Suma, 34
Suma exclusiva, 356

Suma lógica, 356
 switch, 60
 System, 17
 System.Drawing, 434

T

tabla, 113
 Tabla (bases de datos), 289
 Tablas bidimensionales, 122
 Tablas hash, 320
 Tan, 172
 Tangente, 172
 Tanh, 172
 Tao.SDL, 400
 TcpCliente, 350
 Teclado, 340
 Teclado con SDL, 403
 Teclas en consola, 341
 Temporización, 337
 TextBox, 424
 Texto con SDL, 411
 this, 232
 Thread, 338
 Thread.Sleep, 338
 Tipo de datos carácter, 107
 Tipos de datos enteros, 92
 Title, 340
 ToByte, 93
 ToDecimal, 100
 ToInt16, 93
 ToInt32, 43, 93
 ToInt64, 93
 ToLower, 133
 ToSbyte, 93
 ToSingle, 100
 ToString, 103
 ToUInt16, 93
 ToUpper, 133
 true, 111
 Truncate, 271
 try, 88

U

uint, 93

ulong, 93
 UML, 185
 Unicode, 390
 Unidades de disco, 348
 Unidades de medida, 385
 unsafe, 330
 ushort, 93
 using, 44

V

Valor absoluto, 172
 Valor devuelto por una función, 160
 Variables, 36
 Variables globales, 163
 Variables locales, 163
 vector, 113
 Ver resultado de un programa, 33
 verdadero, 111
 virtual, 225
 Visibilidad (POO), 191, 208
 Visual Studio, 420
 VisualStudio, 200
 void, 157

W

WaitForExit, 347
 while, 63, 65
 Windows Forms, 420
 Write (BinaryWriter), 269
 Write (FileStream), 267
 WriteByte, 267
 WriteLine, 17
 WriteLine (ficheros), 243

X

XML, 283
 xor, 356

Y

Y, 52
 Year, 337