

1. Toma de contacto con C#

1.1. Conceptos básicos sobre programación

1.1.1. 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.

Ejercicios propuestos

- (1.1.1.1) Hay fragmentos del anterior programa en código máquina que sí se podrían llegar a entender si los analizas con atención. Con la ayuda de alguna calculadora que sea capaz de convertir de números en el sistema binario a números en el sistema decimal (la propia calculadora de Windows puede ser suficiente), mira cual de los bloques de 4 cifras anteriores equivale al número "2" (el valor que se deseaba guardar) y cual equivale al número "1" (la posición de memoria en la que se deseaba guardar ese valor).

1.1.2. 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 al sistema operativo (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 lo que llamaremos "una clase":

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

Como se puede observar, a medida que los lenguajes evolucionan, los programas sencillos tienden a volverse más complicados, aunque a cambio los nuevos lenguajes son capaces de ayudar al programador en más tareas y de mayor complejidad. Por ejemplo, cuando se creó BASIC, en 1964, sus diseñadores no se

plantearon que un programa pudiera necesitar conectar a Internet, igual que no se consideraba que la facilidad para crear "interfaces gráficas de usuario" fuera algo relevante para un lenguaje cuando se diseñó C en 1972.

Afortunadamente, no todos los lenguajes siguen esta tendencia de avanzar hacia una mayor complejidad, 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("Hola")
```

Y lo mismo ocurre en versiones recientes de C#, que permitirían simplificar nuestro programa hasta llegar a:

```
Console.WriteLine("Hola");
```

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

```
dosseg
.model small
.stack 100h

.data
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 existe 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 en 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. Los colores dependerán del entorno de desarrollo utilizado, y frecuentemente serán personalizables por el usuario).

Ejercicios propuestos

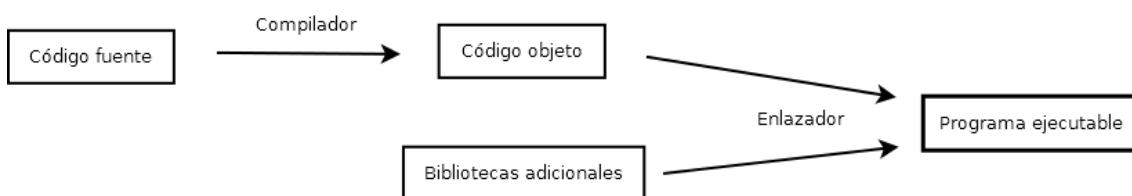
- **(1.1.2.1)** Analiza el programa que está escrito en lenguaje C#. ¿Qué cambio esperas que se deba realizar para escribir "Hasta luego" en vez de "Hola"?

1.1.3. Ensambladores, compiladores e intérpretes

Como hemos visto, las órdenes que nosotros hemos escrito (lo que se conoce como "programa **fuente**") 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 varios pasos: generar un "código máquina" que todavía no será utilizable, quizás recopilar varios fuentes distintos y quizás incluir funcionalidades adicionales que se encuentran en otras bibliotecas que no hemos preparado nosotros. Las herramientas encargadas de realizar todo esto son los **compiladores**.



El programa ejecutable que se ha obtenido con un compilador o un 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 del programa que nos saluda en lenguaje Pascal, tendríamos un fichero fuente llamado SALUDO.PAS. Este fichero inicialmente 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 (Windows, en este caso), 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 otro tipo de traductor, 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.

En principio, es esperable que un intérprete sea más sencillo que un compilador. Por eso eran muy frecuentes en los años 70 u 80, cuando la capacidad de proceso y la memoria disponible en los ordenadores eran muy limitadas.

Aun así, hoy en día los intérpretes siguen siendo muy habituales, especialmente en los entornos en los que es importante responder a las primeras órdenes cuanto antes, sin perder tiempo de analizar todo el programa de principio a fin. 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 para dar lugar a un ejecutable diseñado para un ordenador concreto, sino que se crea 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" (y las hay para la mayoría de sistemas operativos). A cambio, esa "capa intermedia" que supone la máquina virtual se suele traducir en que los programas funcionen algo más despacio que su equivalente compilado para una plataforma específica.

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

- **(1.1.3.1)** 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).
- **(1.1.3.2)** 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).
- **(1.1.3.3)** 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 quizás no vuelvas a utilizar).

- **(1.1.3.4)** 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).

1.1.4. Pseudocódigo y algoritmo

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

La 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 y poderse resolver en un tiempo finito.

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
        En caso contrario
            Avisar de saldo insuficiente
        Fin Si
    En caso contrario
        Avisar de PIN incorrecto
    Fin Si
En caso contrario
    Avisar de fallo de conexión
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

- (1.1.4.1) ¿Qué esperas que escriba en pantalla el siguiente pseudocódigo? Localiza en Internet el intérprete de Pseudocódigo llamado PseInt y prueba a escribirlo y lanzarlo. (**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
```

1.2. ¿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 cerca del año 2000, para realizar programas para su plataforma .NET, pero fue estandarizado posteriormente por ECMA y por ISO, y existe una implementación alternativa de "código abierto", llamada "proyecto Mono", que está disponible para Windows, Linux, Mac OS X y otros sistemas operativos.

Nosotros comenzaremos por usar en los primeros temas el compilador que incorpora la propia plataforma .Net, junto con un editor de texto para programadores. Cuando los conceptos básicos estén asentados, pasaremos a emplear Visual Studio, 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 Community Edition). Existen otros entornos integrados 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.

Ejercicios propuestos

- (1.2.1) Investiga en qué año se crearon los lenguajes C, C++, Java y C#.

1.3. 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:

```
class Ejemplo_01_03a
{
    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 órdenes del sistema (System).
- Las llaves {} se usan para delimitar un bloque de programa. En nuestro caso, se trata del bloque principal del programa (Main).
- `static void Main()` : Main indica cual es "el cuerpo del programa", la parte principal (porque un programa puede estar dividido en varios fragmentos, como veremos más adelante). Todos los programas deben tener un bloque "Main". Los detalles de por qué hay que poner delante "static void" y de por qué se añade después un paréntesis vacío los iremos aclarando más tarde. De momento, deberemos memorizar que ésa será la forma habitual (aunque no la única) de escribir "Main".
- `class Ejemplo_01_03a` : De momento pensaremos que "Ejemplo_01_03a" 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 "class" será obligatorio. Nuevamente, aplazamos para más tarde los detalles sobre qué quiere decir "class".

Nota: en ocasiones verás programas en los que aparece la palabra "**public**" antes de "class" y antes de "static void Main". Ya veremos más adelante por qué, pero uno de esos "public" no es necesario en programas tan sencillos y el otro está incluso desaconsejado por la propia Microsoft en sus últimas recomendaciones. Aun así, será aceptable que aparezcan ambas palabras "public" (o una de ellas), como ocurre en esta variante del ejemplo anterior:

```
public class Ejemplo_01_03a2
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

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 "static", de "void", de "class"... Por ahora nos limitaremos

a utilizar un esqueleto como ese y "rellenar" el cuerpo del programa para entender los conceptos básicos de programación.

Ejercicio propuesto (1.3.1): Crea un programa en C# que te salude por tu nombre (por ejemplo, "Hola, Nacho").

Sólo un par de detalles 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):

```
class Ejemplo_01_03b {
    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:

```
class Ejemplo_01_03c {
    static void Main(){
        System.Console.WriteLine("Hola");
    }
}
```

(esta es la forma que se empleará preferentemente en este texto en el caso de que estemos trabajando con fuentes de mayor tamaño, para que ocupe un poco menos de espacio; en la mayoría de prgoramas 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.

1.4. Cómo probar este programa

Para probar este programa será necesario tener un editor de texto y un compilador de C#, o bien un entorno desarrollo integrado, que incluya ambos. Vamos a ver los pasos requeridos para instalar y usar ambas herramientas, tanto en Linux como en Windows.

1.4.1. Cómo probarlo con un compilador online

Cada vez son más frecuentes sitios web que incluyen "compiladores online", en los que podemos teclear nuestro programa y probarlo, sin necesidad de instalar nada en nuestro equipo ni de más herramientas que un navegador web.

Uno de ellos es "repl.it", que no necesita registro y muestra un panel izquierdo en el que podemos teclear nuestro programa y un panel derecho en el que, si pulsamos el botón "run", podemos ver su resultado:

The screenshot shows a web browser window for repl.it. The address bar says 'Es seguro | https://repl.it/repls/ComplexEuphoricAutotote'. The main interface has a left sidebar with a file named 'main.cs' containing the following C# code:

```

1  public class Ejemplo01
2  {
3      public static void Main()
4      {
5          System.Console.WriteLine("Hola");
6      }
7 }

```

Next to the code is a green 'run ▶' button. To the right is a terminal window titled 'Mono C# compiler version 4.0.4.0' which displays the output: 'Hola'.

Esta forma de trabajar tendrá dos problemas:

- No será adecuada para proyectos grandes, que estén formados por varios ficheros.

- Por lo general, no será posible "depurar el programa" (avanzar paso a paso, ver valores de variables, y una serie de operaciones avanzadas que veremos más adelante).

Un tercer problema que encontraremos en muchos entornos online es que no se podrán usar con comodidad para programas interactivos, que respondan a datos introducidos por el usuario. Algunos de ellos, como el de "repl.it", sí permitirán introducir datos de forma sencilla.

Ejercicio propuesto (1.4.1.1): Usa el compilador online de "repl.it" (o algún otro sitio web similar) para crear y probar un programa en C# que escriba en pantalla "Bienvenido a C#".

1.4.2. 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.

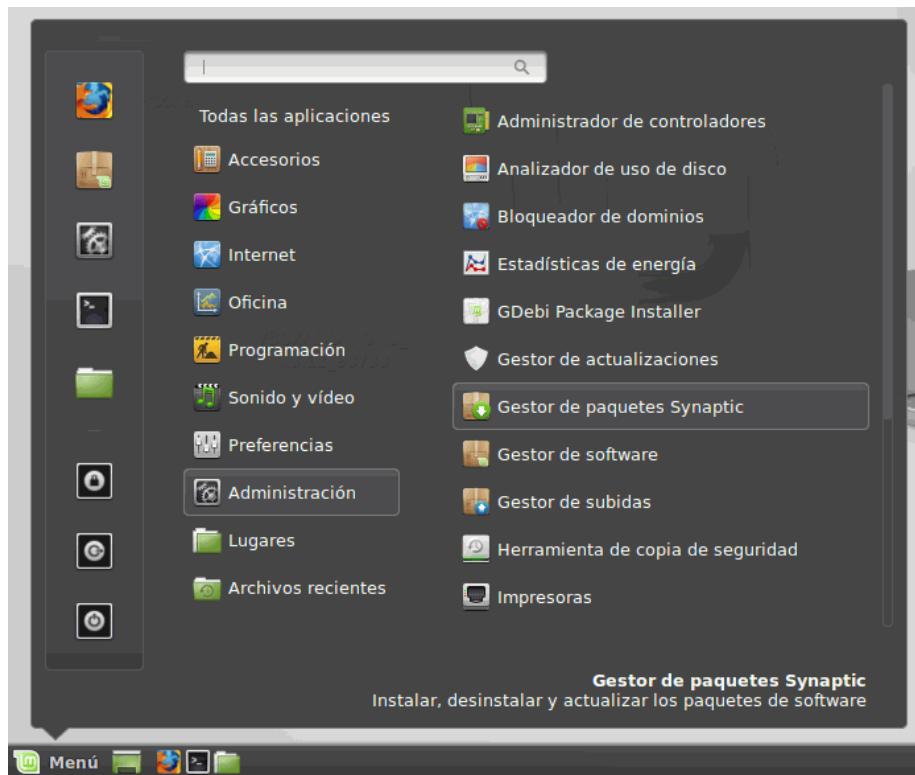
La instalación de Linux, que podría tener la dificultad de crear particiones para coexistir con nuestro sistema operativo habitual, hoy en día puede realizarse de forma simple, usando software de virtualización gratuito, como VirtualBox, que permite tener un "**ordenador virtual**" dentro del nuestro, 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 como entorno de desarrollo (en su versión 17 Cinnamon, pero los cambios deberían ser mínimos para versiones posteriores):

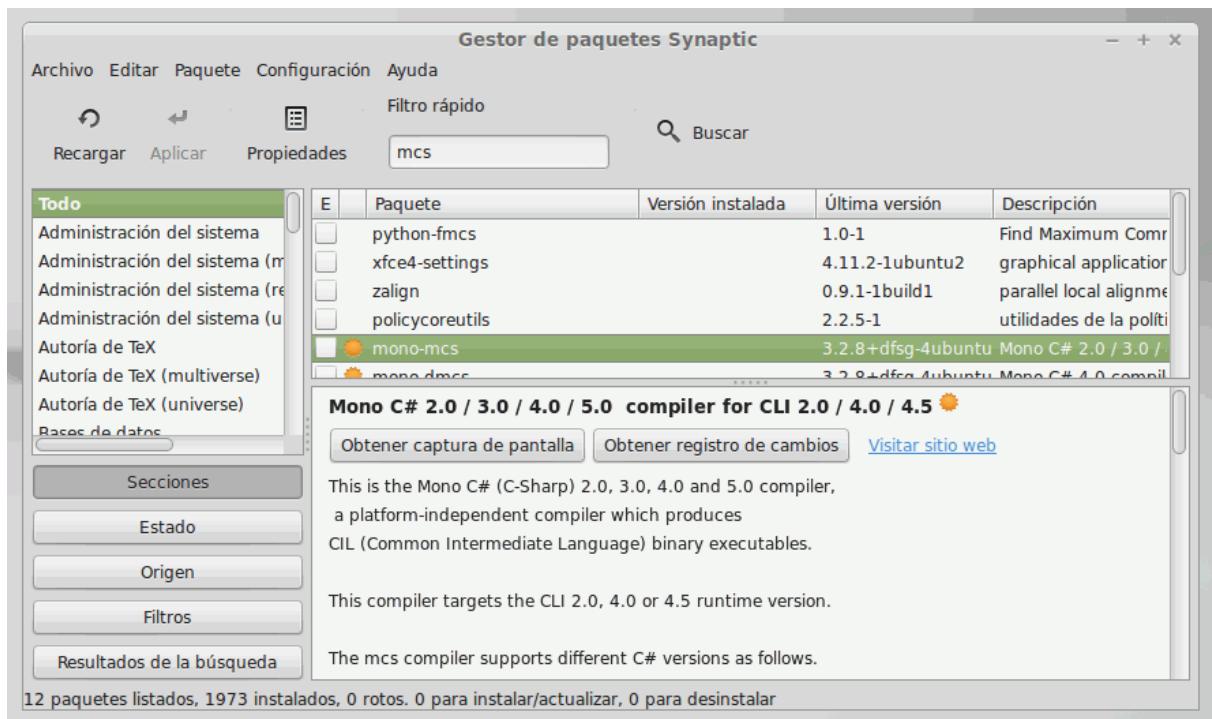


En primer lugar, deberemos entrar al instalador de software de nuestro sistema, que para las versiones de Linux Mint basadas en escritorios derivados de Gnome (como es el caso de Mint 17 Cinnamon), suele ser un tal "Gestor de paquetes Synaptic":

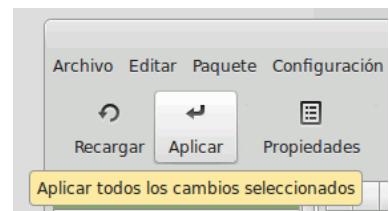


Se nos pedirá nuestra contraseña de usuario (la que hayamos utilizado en el momento de instalar Linux), y aparecerá la pantalla principal de Synaptic, con una enorme lista de software que podríamos instalar. En esta lista, aparece una casilla

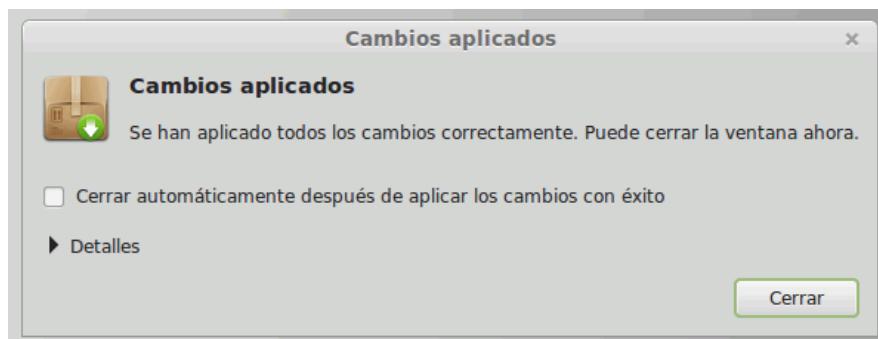
de texto llamada "Filtro rápido", en la que podemos teclear "mcs" para que nos aparezca directamente nuestro compilador Mono:



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 sea necesario instalar algún otro paquete adicional y entonces ya podremos pulsar el botón "Aplicar":

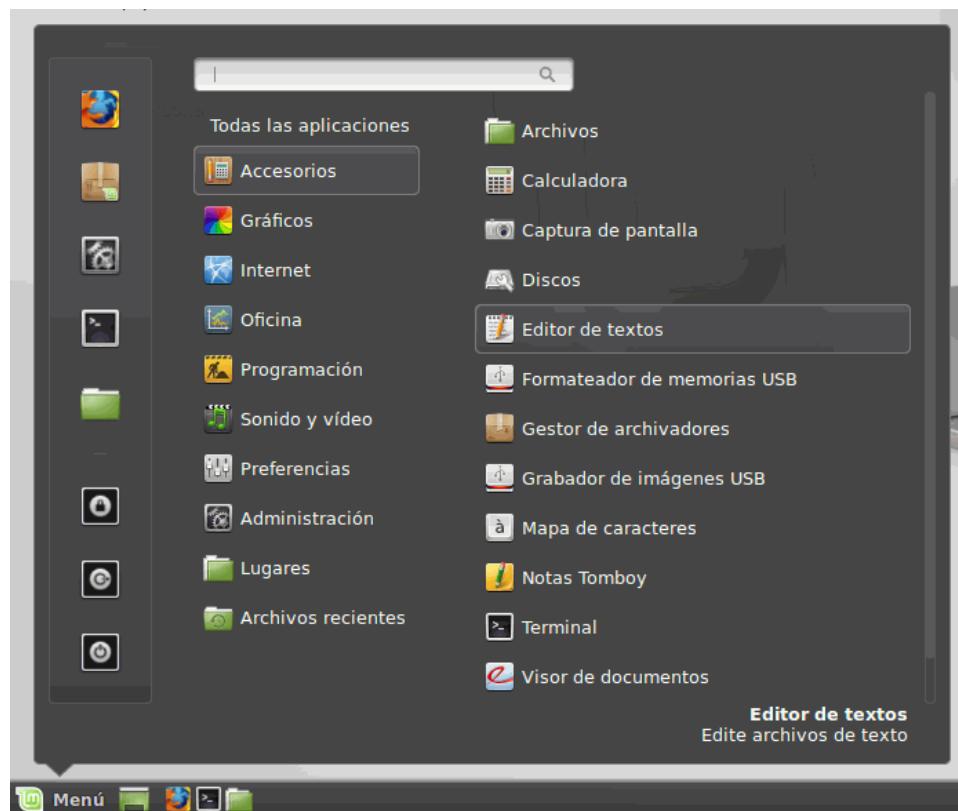


Se descargarán los ficheros necesarios, se instalarán y al cabo de un instante se nos avisará de que se han aplicado todos los cambios:

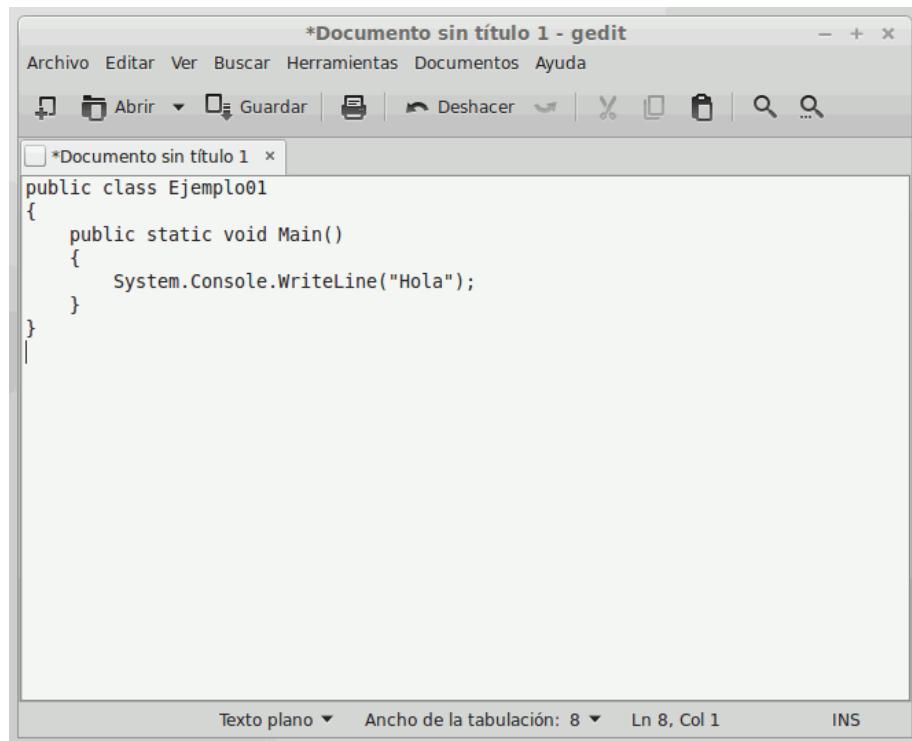


Ya tenemos instalado el compilador, que es la herramienta que convertirá nuestros programas en algo que el ordenador realmente entienda. Para teclear los programas necesitaremos un editor de texto, pero eso es algo que viene

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:



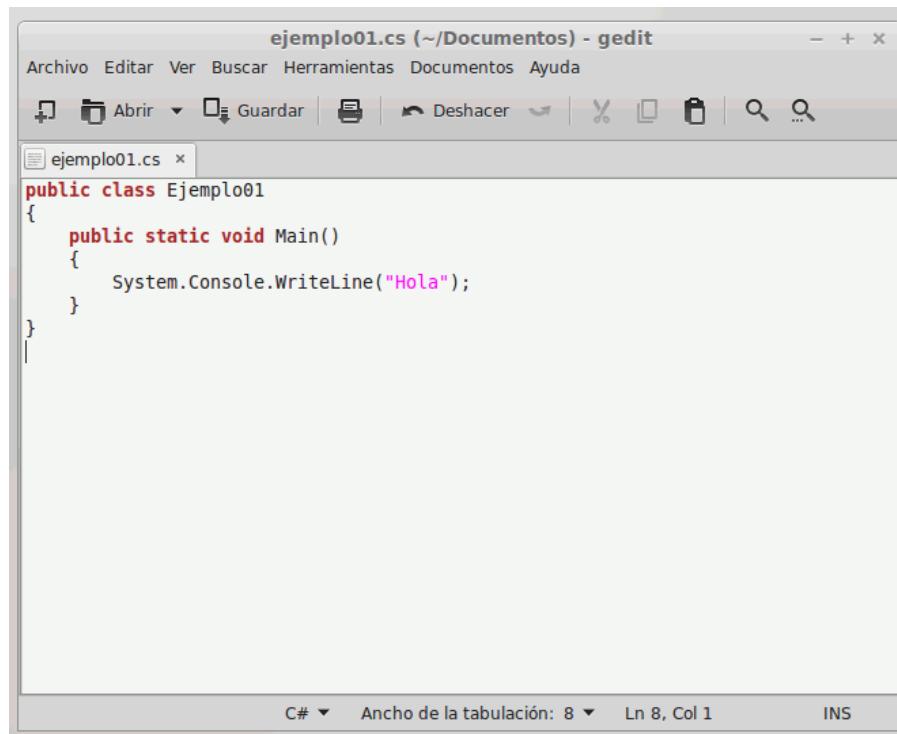
```

*Documento sin título 1 - gedit
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Abrir Guardar Deshacer | X | 
*Documento sin título 1 x
public class Ejemplo01
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}

```

Texto plano ▾ Ancho de la tabulación: 8 ▾ Ln 8, Col 1 INS

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:



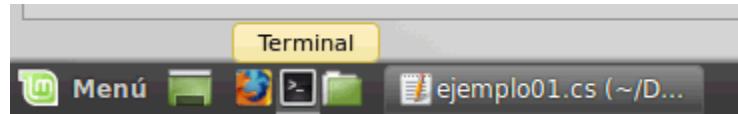
```

ejemplo01.cs (~/Documentos) - gedit
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Abrir Guardar Deshacer | X | 
ejemplo01.cs x
public class Ejemplo01
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}

```

C# ▾ Ancho de la tabulación: 8 ▾ Ln 8, Col 1 INS

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:

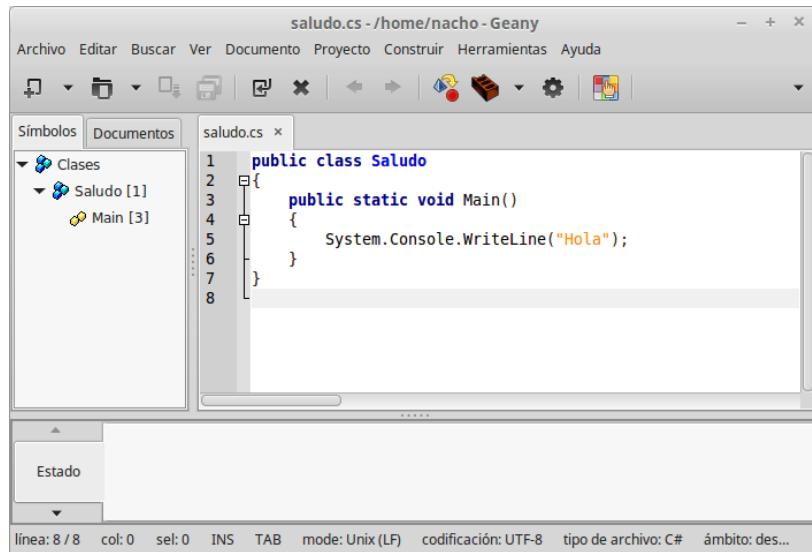
- Si hemos guardado el fuente en la carpeta "Documentos", el primer paso será entrar a esa carpeta con la orden "cd Documentos".
- Despues lanzaremos el compilador con la orden "mcs" seguida del nombre del fuente: "mcs ejemplo01.cs" (recuerda que en Linux debes respetar las mayúsculas y minúsculas tal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

```
nacho@nacho-mint17 ~ $ cd Documentos/
nacho@nacho-mint17 ~/Documentos $ mcs ejemplo01.cs
nacho@nacho-mint17 ~/Documentos $ mono exemplo01.exe
Hola
nacho@nacho-mint17 ~/Documentos $
```

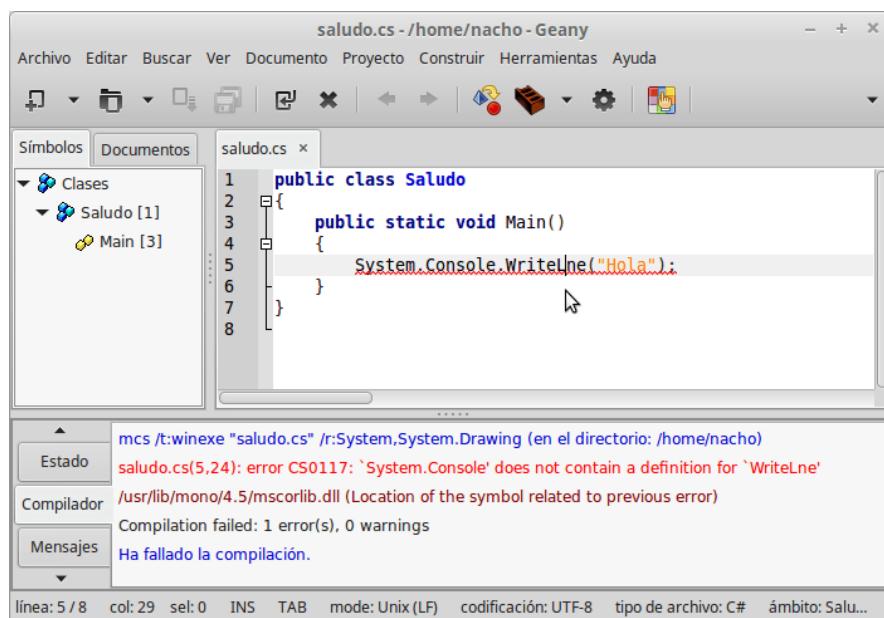
Si obtienes algún mensaje de error, tendrás que comprobar si has dado los pasos anteriores de forma correcta y si tu fuente está bien tecleado.

Ejercicio propuesto (1.4.2.1): Si vas a utilizar Linux, instala el software de desarrollo y despues crea y prueba un programa en C# que escriba en pantalla "Bienvenido a Linux".

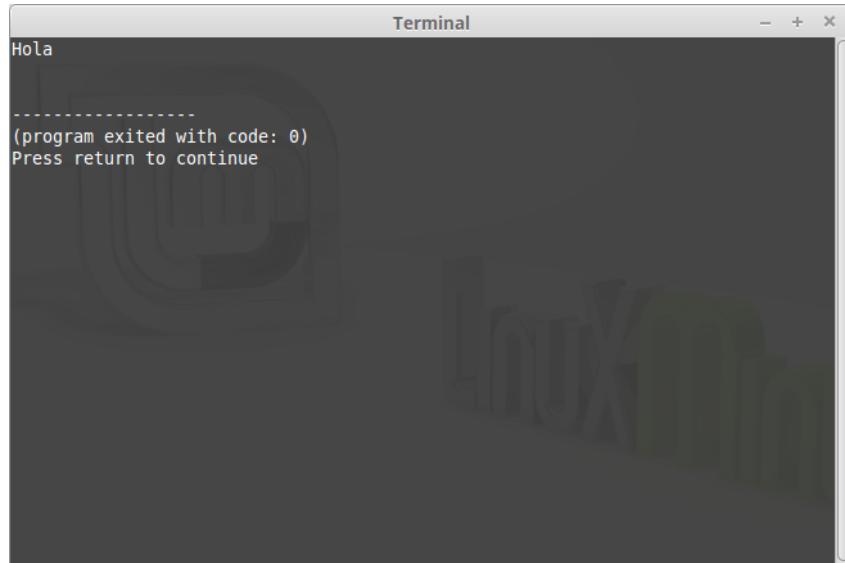
Una alternativa aún más cómoda es no abrir una consola para compilar el fuente y probar el resultado, sino utilizar un editor un poco más avanzado, que permita realizar ambas tareas desde el propio editor. Uno de los que lo incluye estas posibilidades (y que además es muy ligero) es **Geany**, que no viene preinstalado en la mayoría de distribuciones de Linux, pero se puede instalar en pocos segundos empleando Synaptic o el gestor de paquetes de nuestro sistema.



En Geany encontrarás un botón "Compilar", que lanza el compilador por ti, e incluso destaca en color rojo las líneas que contengan algún error:



y también tiene un botón "Ejecutar", que lanza el programa y espera a que se pulse una tecla antes de volver al editor:



Ejercicio propuesto (1.4.2.2): Si vas a utilizar Linux, instala también Geany y crea y prueba un programa en C# que escriba en pantalla "Bienvenido a Linux y a Geany".

1.4.3. Windows y equipos potentes: Visual Studio

Si usas Windows y tienes un equipo reciente (al menos 4 GB de memoria RAM - recomendable 8 GB o más- y al menos 2.0 GHz de velocidad de procesador de 64 bits -recomendable quad core o más-), la alternativa más profesional es utilizar Visual Studio, de Microsoft. La versión "Community" de Visual Studio es gratuita para educación y para desarrolladores individuales:

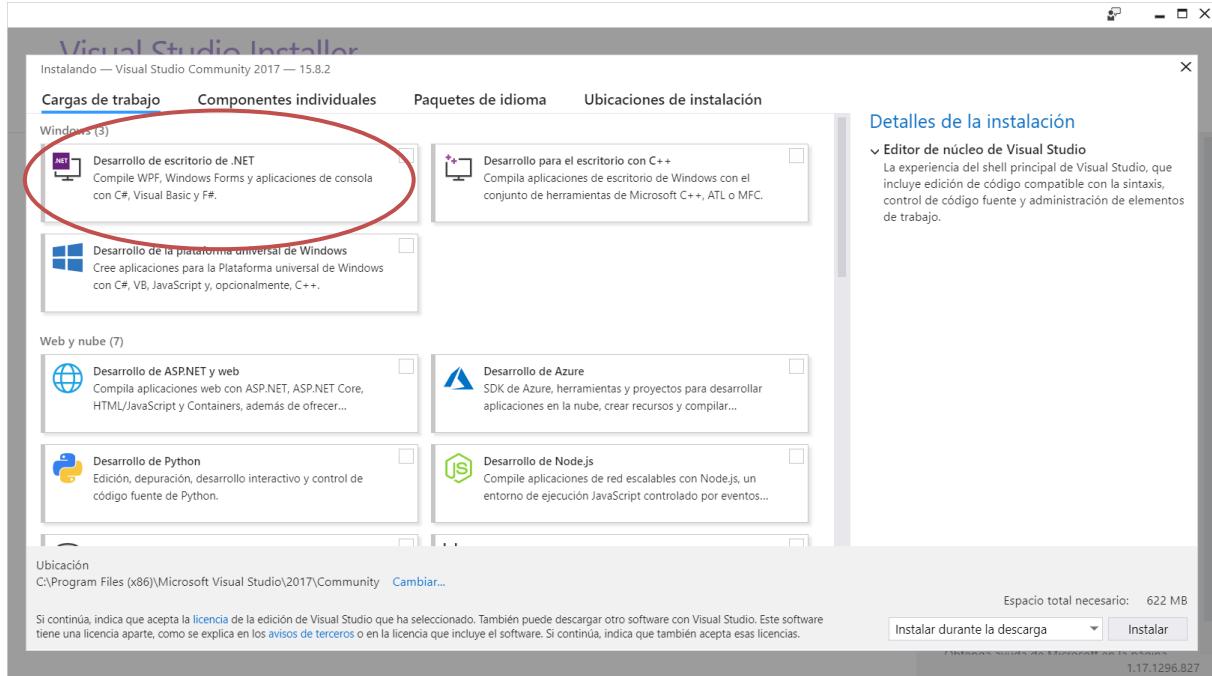
<https://visualstudio.microsoft.com/es/vs/community/>

(Si tu centro de estudios es parte del programa Microsoft para estudiantes, quizás eso te dé acceso gratuito a las versiones Professional y/o Enterprise, pero éstas consumen muchos más recursos y no aportan gran cosa para un principiante, por lo que en principio son menos recomendables para seguir este texto).

En versiones anteriores a la 2017, era posible descargar una imagen ISO de un DVD, para poder instalar Visual Studio en equipos en los que la conexión de Internet sea lenta o tenga un límite de datos. En la versión 2017 y posteriores, por el contrario, sólo existe un instalador online¹, pero, a cambio, éste permite descargar sólo las herramientas que realmente vayamos a utilizar.

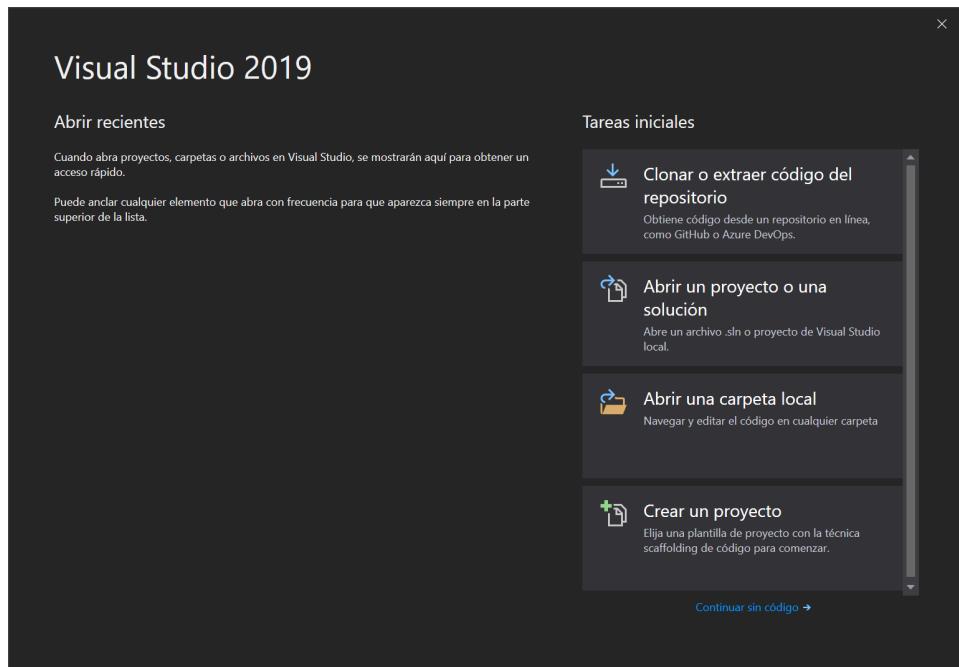
¹ Realmente, es posible crear un "instalador offline" para Visual Studio 2017 y posteriores, descargando los ficheros necesarios mediante el instalador predeterminado, y luego se podrían llevar todos esos ficheros a

El primer paso de la instalación será escoger qué herramientas deseamos instalar. Para el propósito de este texto nos bastaría con la primera opción, "Desarrollo de escritorio .Net":

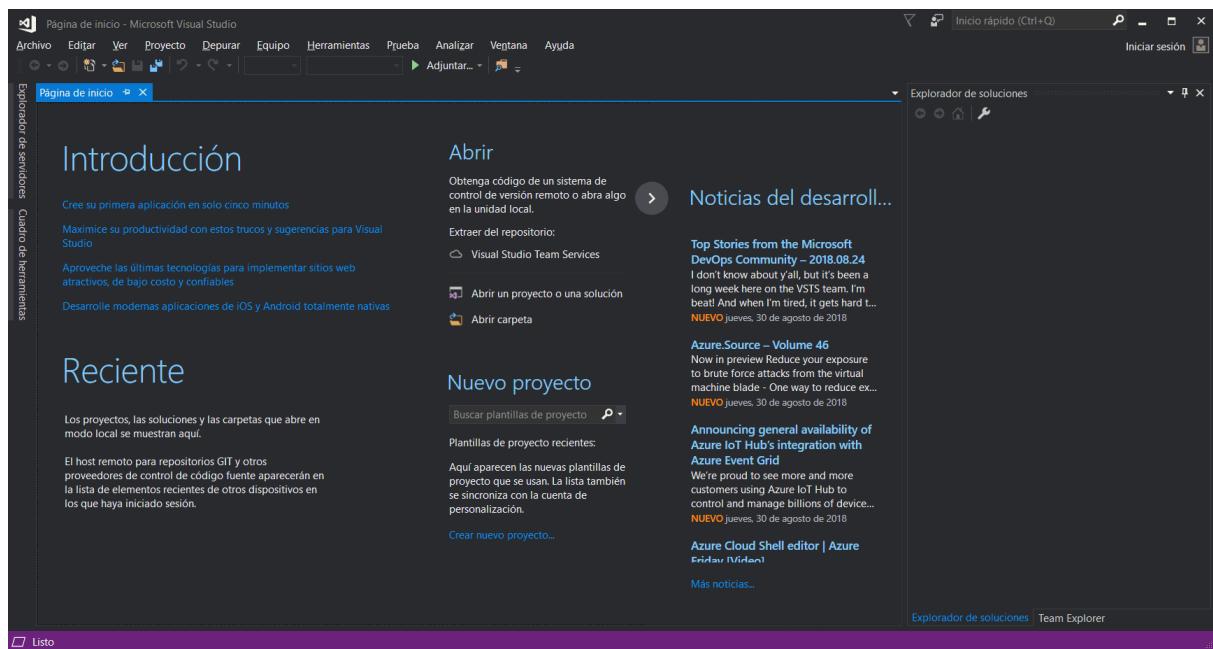


Cuando lo instales, verás una pantalla como ésta (la combinación de colores puede ser distinta en tu caso):

cualquier otro equipo, aunque no tuviera conexión a Internet. En cualquier caso, se trata de un proceso relativamente engorroso, por lo que no entraremos en más detalle.



O, en versiones más antiguas, como Visual Studio 2017, quizá sea como ésta:

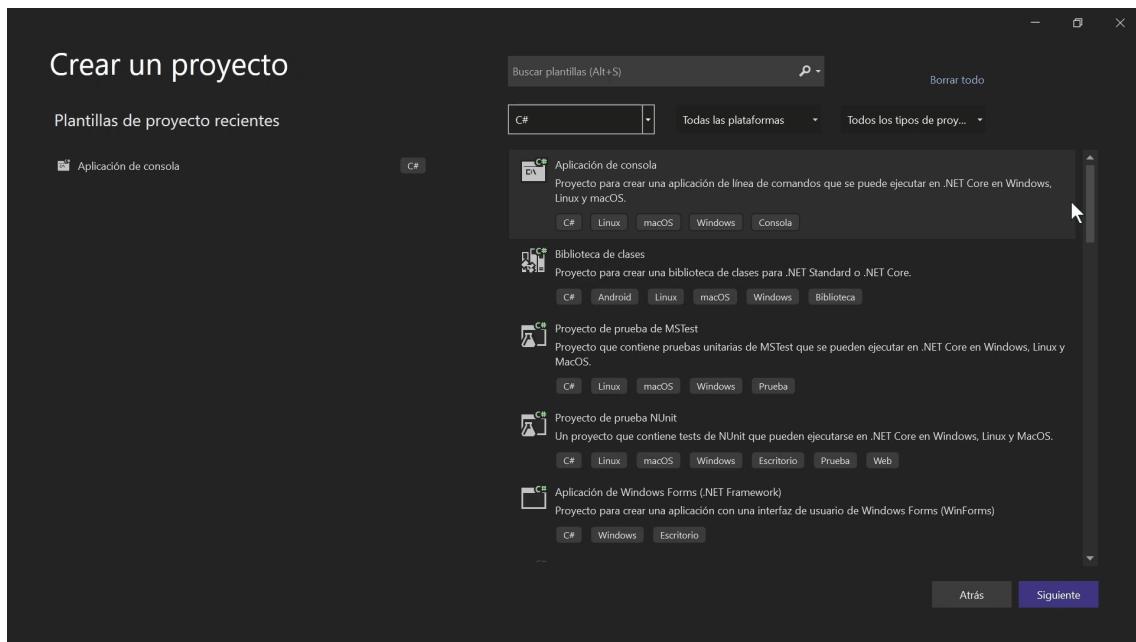


Cuidado: es probable que tengas un **mes de prueba** inicial. A partir de ese momento, Visual Studio Community sigue siendo gratis, pero quizás tengas que registrar tu copia para que siga activada:



Para crear un programa, deberás comenzar por crear un proyecto, bien desde la correspondiente opción de la pantalla principal, o bien desde Archivo, en la opción "Nuevo / Proyecto".

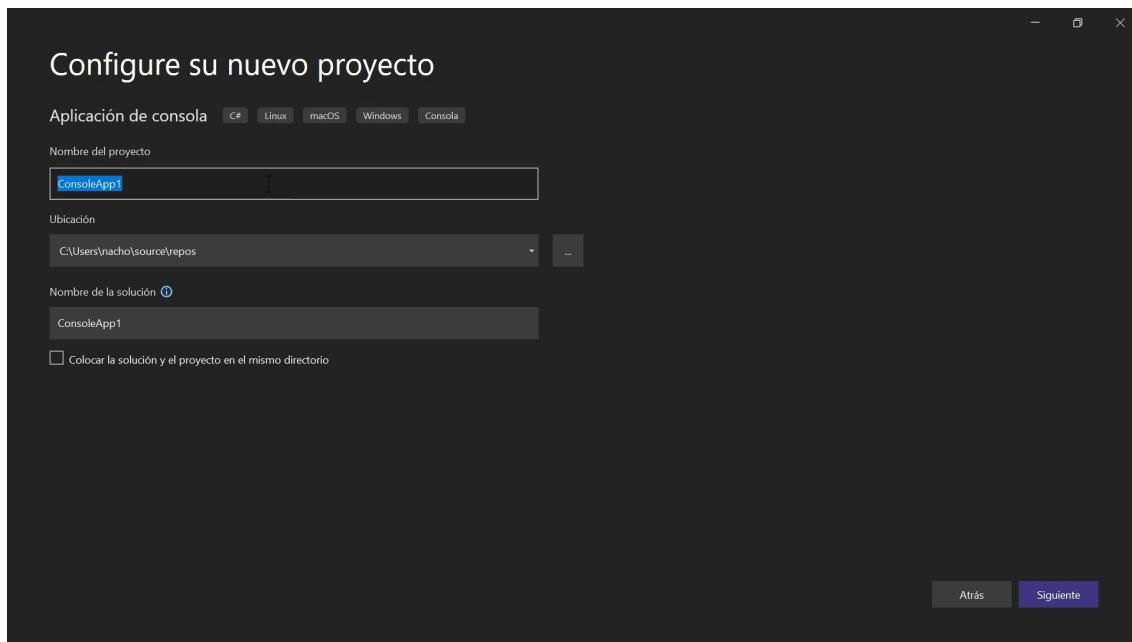
Entre los tipos de proyectos disponibles, los que deberás comenzar realizando como aprendiz de programación serán "Aplicación de consola":



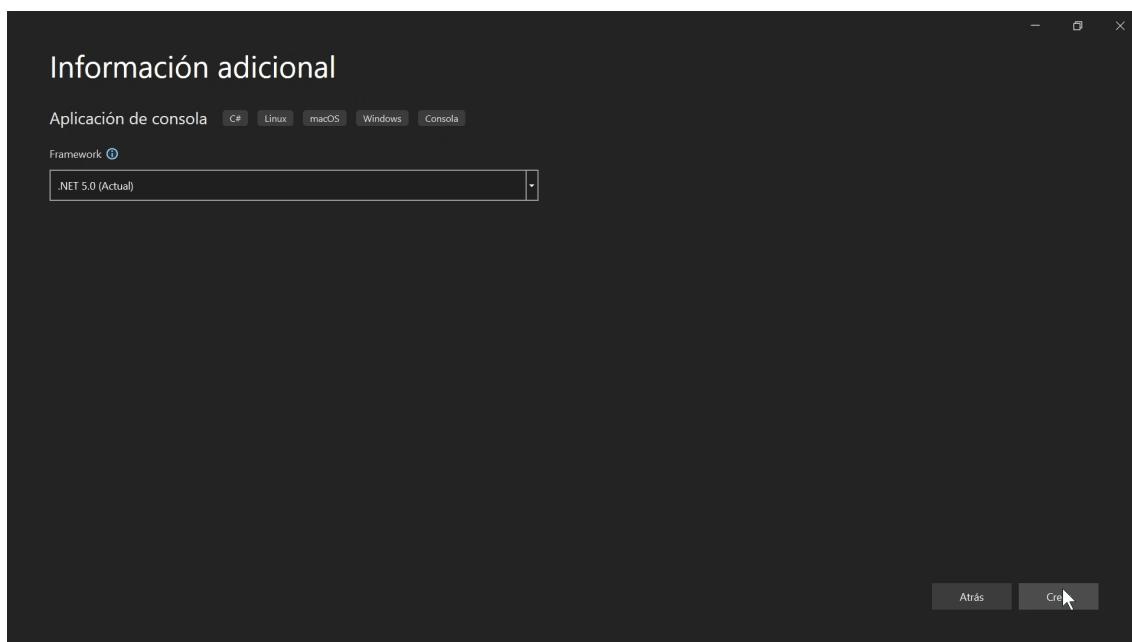
Se te propondrá el nombre `ConsoleApp1`, que deberás cambiar por otro nombre que indique la misión del programa que estás creando (en este ejemplo, "Saludo").

También es importante que te fijes en la casilla "ubicación", que te indica en qué carpeta de tu ordenador se va a guardar tu programa. Necesitarás conocer esa carpeta si se trata de un programa que debas entregar.

En el caso de **Visual Studio 2022**, en un primer paso se te pedirá el nombre y la ubicación:

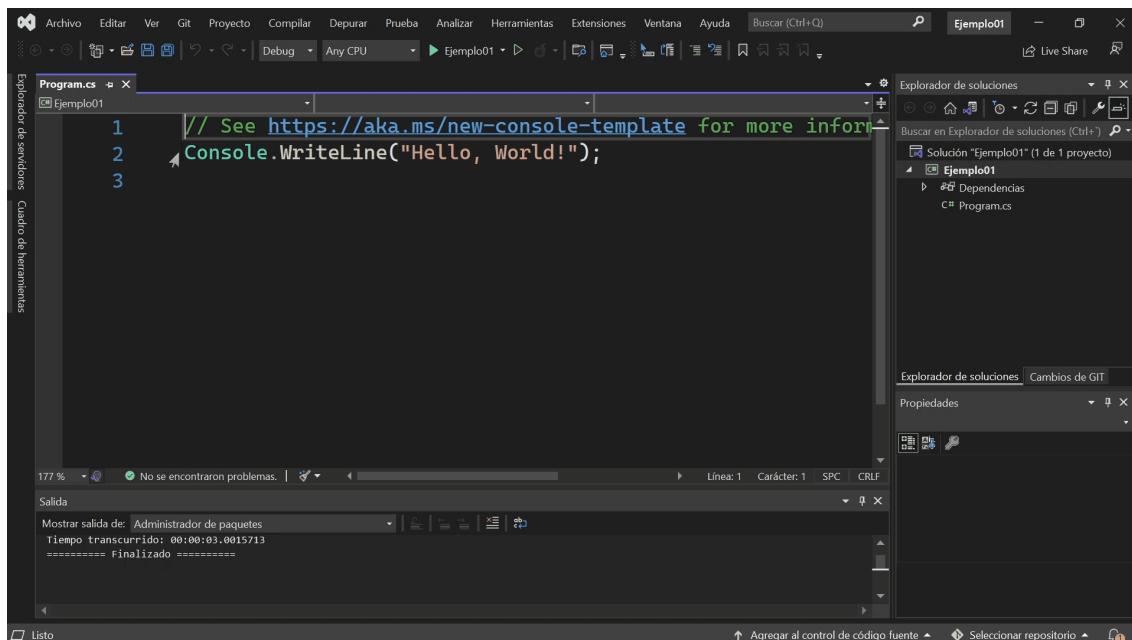


Y en un segundo paso se te preguntará la plataforma de destino. Por motivos que veremos enseguida, será recomendable escoger la plataforma ".Net 5.0" o una versión anterior:

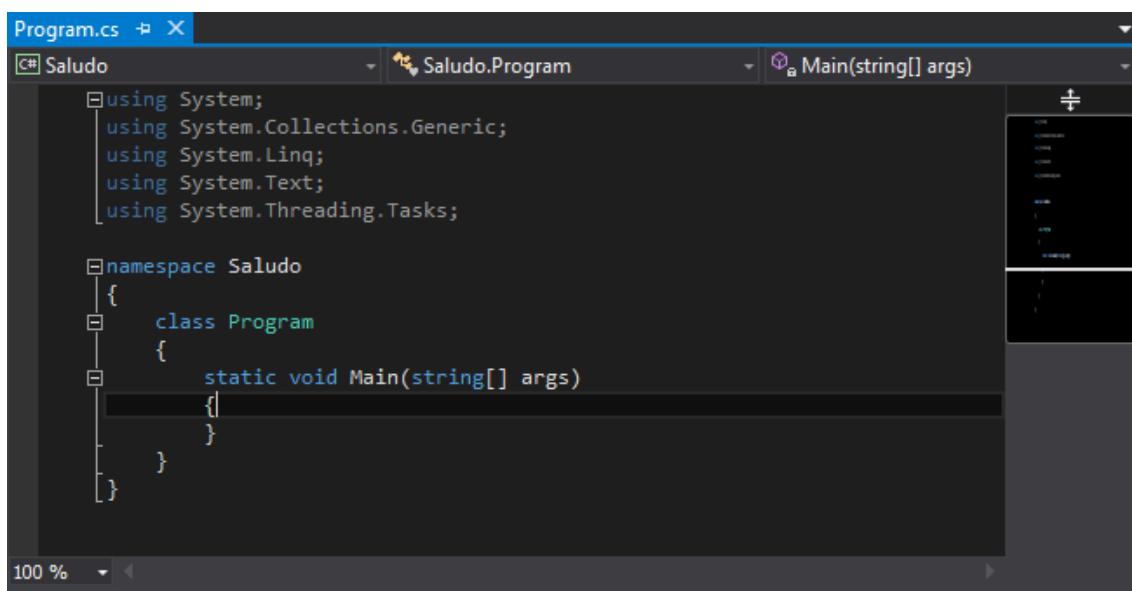


El problema de escoger una versión más moderna (la 6.0, en este momento) es que el programa estará más simplificado, llegando casi al nivel de Python, lo que

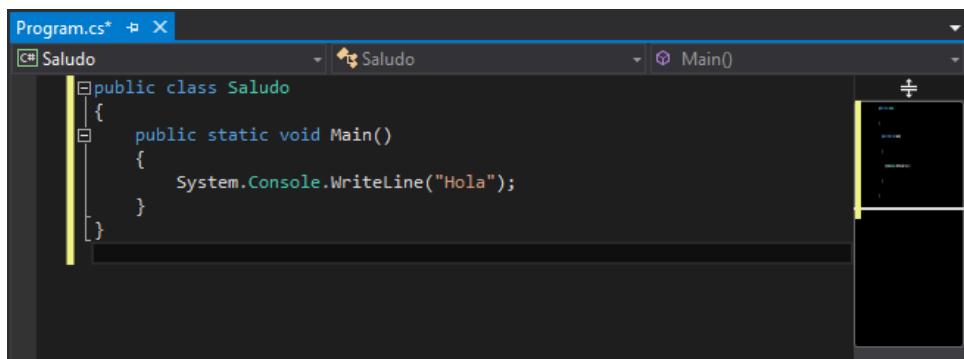
permitiría esquivar ciertos detalles que un programador "de carrera" debe conocer, así que trabajaremos con un esqueleto de programa "más convencional".



Si has escogido la versión 5.0 o anterior, aparecerá un esqueleto de programa en el que **"sobran cosas"**: varias líneas **"using"**, que habría que borrar (en un programa normal sólo conservaremos una, como veremos dentro de poco), así como un **"namespace"**, que ayuda a evitar colisiones de nombres en proyectos de gran tamaño pero que será innecesario en programas tan sencillos:



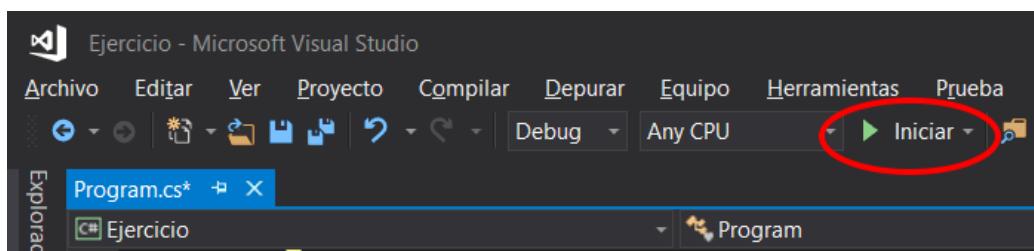
La apariencia real de tu programa finalizado debería ser algo como:



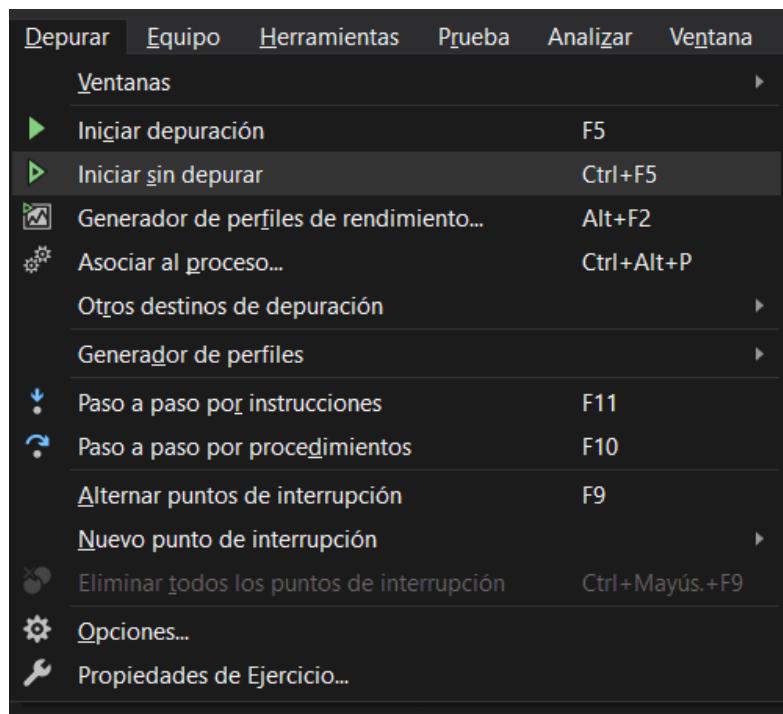
```
Program.cs*  X
# Saludo
public class Saludo
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

(tras eliminar el "namespace", si tu programa queda demasiado a la derecha, puedes desplazar todo el cuerpo del programa un poco más a la izquierda, seleccionándolo y pulsando Mayús+Tab).

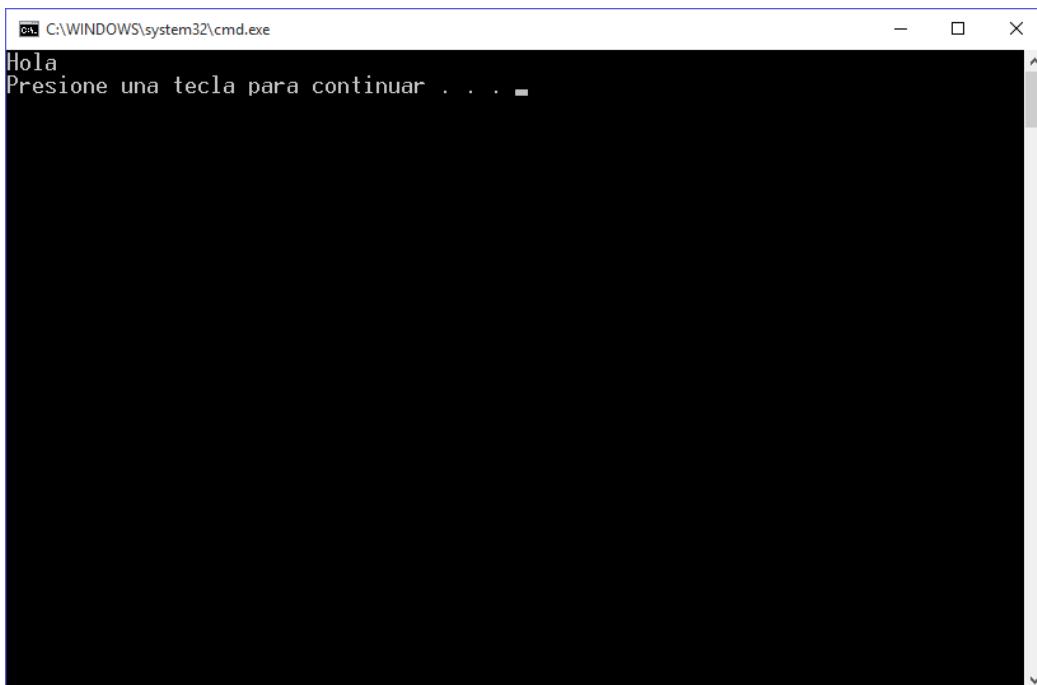
Para **lanzar** un programa, puedes usar el botón "Iniciar", o pulsar la tecla F5, o la correspondiente opción del menú Depurar:



Pero esa opción tiene un problema en los programas tan sencillos y no interactivos: el programa se lanza y **la ventana de ejecución se cierra** inmediatamente, por lo que probablemente no tendrás tiempo de comprobar los resultados. La alternativa es pulsar **Ctrl+F5** o usar la opción "Iniciar sin depurar" del menú Depurar:



De modo que se podrá comprobar el resultado, que sería:



Otra **alternativa** (poco elegante y que deberías intentar evitar), si usas una versión de Visual Studio antigua, que no permita pulsar Ctrl+F5 para hacer una pausa al final de la ejecución, es añadir una orden "ReadLine", de modo que sea tu propio programa el que se encargue de esperar a que el usuario pulse Intro:

```

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

```

Eso sí, esa línea "ReadLine" no debería ser parte de un programa definitivo, y **deberías borrarla** antes de entregar un ejercicio que la contenga.

Visual Studio es muy cómodo para el trabajo diario en proyectos grandes, especialmente por detalles que veremos más adelante, como el autocompletado y la facilidad para depurar, pero tiene un par de **inconvenientes**:

- Requiere bastantes recursos: Ocupa mucho espacio en disco, consume mucha memoria durante su funcionamiento y necesita un procesador relativamente potente.
- Hay que crear un nuevo proyecto para cada fuente, lo que puede suponer tener decenas de carpetas si haces todos los ejercicios propuestos (y sí, deberías hacer todos ellos si quieras asegurarte de asentar tus conocimientos).

Ejercicio propuesto (1.4.3.1): Si vas a emplear Visual Studio, instálalo y después crea y prueba un programa en C# que escriba en pantalla "Bienvenido a Visual Studio" (recuerda pulsar Ctrl+F5 para que se haga una pausa al final del programa sin necesidad de emplear la orden ReadLine).

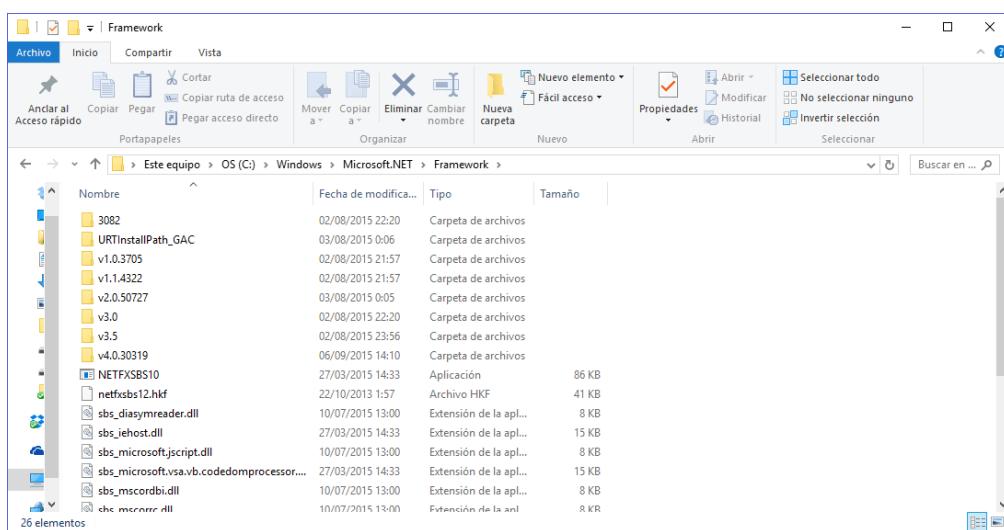
1.4.4. Windows y equipos poco potentes: .Net y Geany

Si tu equipo informático no es muy reciente, o si prefieres un entorno de desarrollo más ligero, hay una alternativa relativamente sencilla y bastante parecida a la que hemos visto para Linux: instalar Geany y utilizar el compilador de C# que es probable que ya tengas en tu sistema. Ésta es la alternativa recomendada para los primeros temas de este curso (1 al 5).

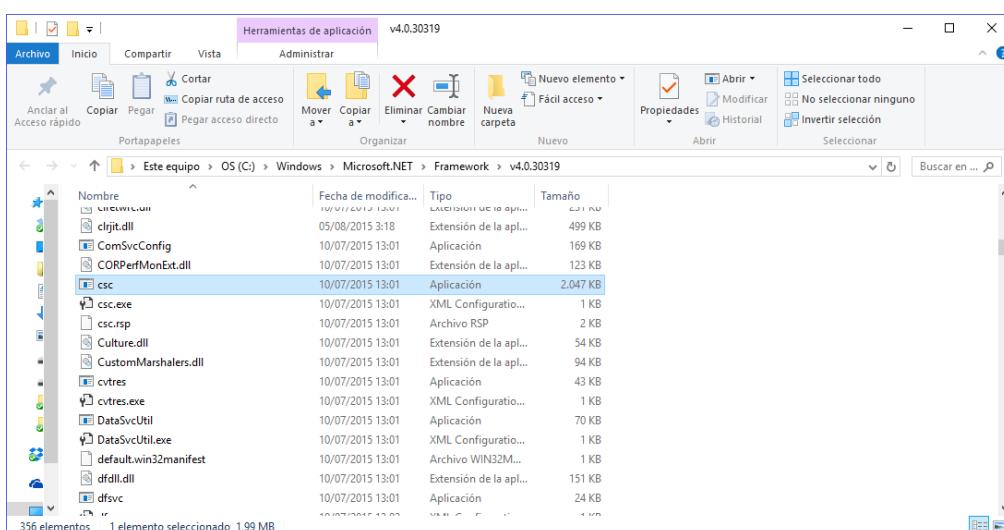
La plataforma .Net, en una u otra versión, viene preinstalada en los equipos con versiones de Windows posteriores a XP (e incluso en Windows XP, si has instalado

los "Service Pack"). Es más, esta plataforma no incluye sólo lo necesario para utilizar programas creados en C#, sino también un compilador para crearlos.

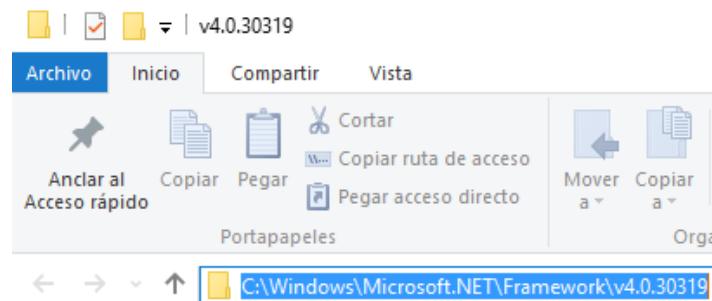
Para encontrar el compilador, deberás abrir un Explorador de archivos, y pasear por la carpeta Windows de tu equipo, subcarpeta "Microsoft .Net", donde habrá una carpeta "Framework" y, si el sistema operativo es de 64 bits, una "Framework64". En ellas existirán distintas subcarpetas, una para cada versión de la plataforma .Net que tengamos instalada (por ejemplo, "v2.0.50727" para la versión 2 y "v4.0.300319" para la versión 4).



En algunas de esas carpetas existirá un fichero llamado "csc" (de tipo "Aplicación"), que es el compilador de C#:

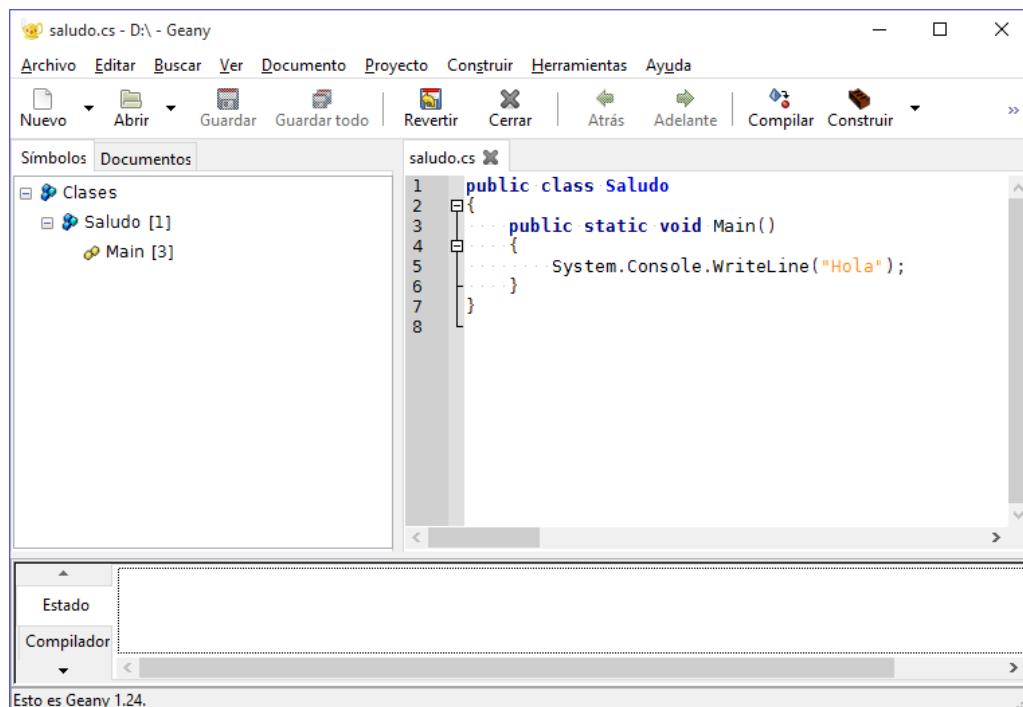


Nos interesará "memorizar esa carpeta". La forma más sencilla posiblemente será hacer clic en la barra de direcciones para ver su ruta completa y luego pulsar Ctrl+C para "copiar" esa dirección:



Si hemos encontrado el compilador, podremos instalar **Geany para Windows** y configurarlo, siguiendo los siguientes pasos:

- Descargar Geany desde www.geany.org e instalarlo.
- Teclear un programa en C# y guardarla con un nombre que termine en ".cs" (a partir de entonces, su sintaxis se verá destacada con colores, como hemos visto para el caso de Linux).

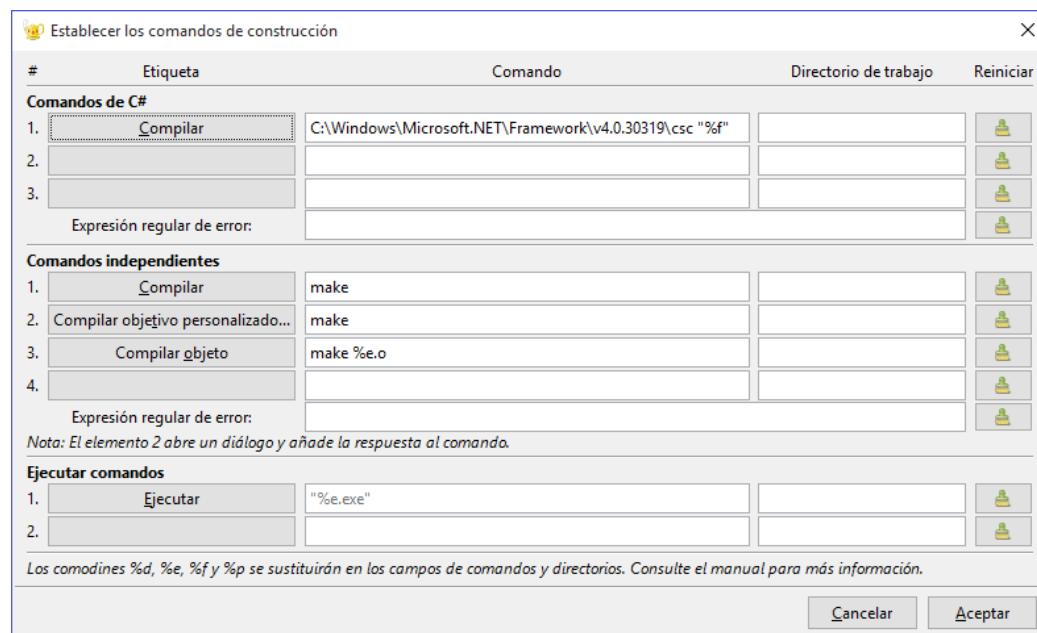


- A continuación, entraremos al menú "Construir" y la opción "Establecer comandos de construcción". En la casilla "Compilar", deberemos "pegar" la

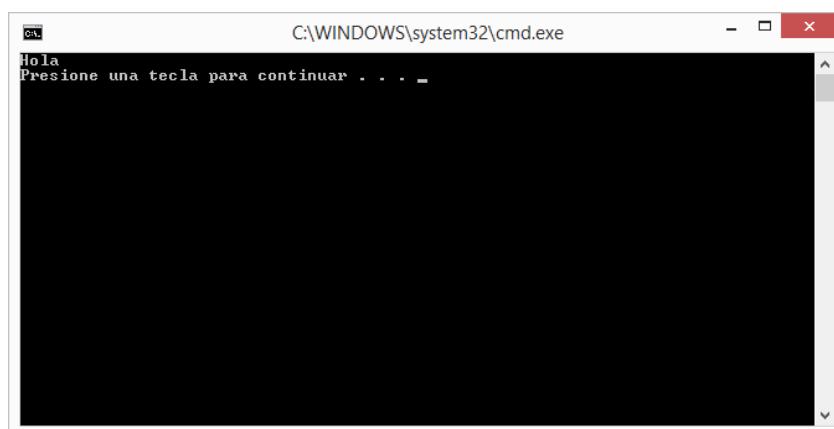
ruta en la que estaba el compilador, terminada en "\csc" y luego, entre comillas, "%f" (que es el símbolo que usa Geany para referirse al nombre del fichero actual), de modo que quedaría algo como

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe "%f"
```

De igual modo, en la casilla Ejecutar debería aparecer algo como "%e.exe" (incluyendo las comillas, para que se comporte correctamente incluso si el fuente está en una carpeta cuyo nombre contenga espacios):



- A partir de ese momento, ya podremos usar el botón Compilar y el botón Ejecutar para generar el ejecutable de nuestro programa y para lanzarlo (con una pausa automática al terminar), como vimos en el caso de Linux.



Ejercicio propuesto (1.4.4.1): Si vas a utilizar Geany, descárgalo y configúralo. Después crea y prueba un programa en C# que escriba en pantalla "Bienvenido a C#".

1.5. Mostrar números enteros en pantalla

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

El ejemplo más sencillo es el de una operación matemática. La forma de realizarla es 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:

```
class Ejemplo_01_05a
{
    static void Main()
    {
        System.Console.WriteLine(3+4);
    }
}
```

Ejercicios propuestos:

- (1.5.1) Crea un programa que diga el resultado de sumar 118 y 56.
- (1.5.2) 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, ahora que todavía estás empezando).

1.6. Operaciones aritméticas básicas

1.6.1. Operadores

Parece evidente que el símbolo de la suma será un +, y podemos esperar cual será el de la resta, pero algunas 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:

```
class Ejemplo_01_06_01a
{
    static void Main()
    {
        System.Console.WriteLine("El resto de dividir 19 entre 5 es");
        System.Console.WriteLine(19 % 5);
    }
}
```

(Nota: como posiblemente imaginas y como podrás haber comprobado con el ejemplo anterior, es habitual que un programa esté formado por más de una orden, y en ese caso se analizarán de la primera -la que está más arriba- a la última -la que se encuentra más abajo-, de una en una).

Por ahora nos centraremos en los números enteros. Verás que el resultado de la división de dos números enteros es otro número que tampoco tiene cifras decimales. Por ejemplo, 15/2 dará como resultado 7. Si quisieras realizar una división con decimales, alguno de los dos operandos deberá tener decimales, aunque no los necesite realmente, como en 15.0/2 o en 15/2.0.

Ejercicios propuestos:

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

1.6.2. Orden de prioridad de los operadores

Debería resultar sencillo porque coincide con la prioridad habitual en matemáticas:

- 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).

```
class Ejemplo_01_06_02a
{
    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: Calcula (a mano y después comprueba desde C#) el resultado de las siguientes operaciones:

- (1.6.2.1) Calcula el resultado de $-2 + 3 * 5$
- (1.6.2.2) Calcula el resultado de $(20+5) \% 6$
- (1.6.2.3) Calcula el resultado de $15 + -5*6 / 10$
- (1.6.2.4) Calcula el resultado de $2 + 10 / 5 * 2 - 7 \% 1$

1.6.3. 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":

```
class Ejemplo_01_06_03a
{
    static void Main()
    {
        System.Console.WriteLine(10000000*10000000);
    }
}
```

1.7. 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 daremos 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.

1.7.1. Definición de variables: números enteros

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

El primer tipo de datos que usaremos serán números enteros (sin cifras 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 (1.7.1.1): Amplía el "Ejemplo 01.06.02a" para declarar tres variables, llamadas n1, n2, n3.

1.7.2. 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 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 propuestos:

(1.7.2.1) Amplía el ejercicio 1.7.1.1, para que las tres variables n1, n2, n3 estén declaradas en la misma línea y tengan valores iniciales.

(1.7.2.2) Amplía el ejercicio 1.7.2.1, declarando también una variable "suma" y guardando en ella el resultado de sumar n1, n2 y n3.

1.7.3. 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:

```
class Ejemplo_01_07_03a
{
    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:

- (1.7.3.1) Crea un programa que calcule el producto de los números 121 y 132, usando variables.
- (1.7.3.2) Crea un programa que calcule la suma de 285 y 1396, usando variables.
- (1.7.3.3) Crea un programa que calcule el resto de dividir 3784 entre 16, usando variables.
- (1.7.3.4) Amplía el ejercicio 1.7.2.1, para que se muestre el resultado de la operación $n1+n2*n3$.
- (1.7.3.5) Amplía el ejercicio 1.7.2.2, para que se muestre la suma de los tres números.

1.8. 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 eñe 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 eñe)
MásDatos	(tiene una vocal acentuada)

(Nota: algunos entornos de programación modernos sí permitirán variables que contengan eñe 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 con todas las 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:

- (1.8.1)** Crea un programa que calcule el producto de los números 87 y 94, usando variables llamadas "numero1" y "numero2".
- (1.8.2)** 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".
- (1.8.3)** 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".
- (1.8.4)** Crea una nueva versión del programa que calcula el producto de los números 87 y 94, usando esta vez variables llamadas "número1" y "número2".

1.9. 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 ---- */

class Ejemplo_01_09a
{
    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 ----

class Ejemplo_01_09b
{
    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:

(1.9.1) 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.

1.10. 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
class Ejemplo_01_10a
{
    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:

- (1.10.1) Crea un programa que calcule el producto de dos números introducidos por el usuario.
- (1.10.2) Crea un programa que calcule la división de dos números introducidos por el usuario, así como el resto de esa división.
- (1.10.3) Suma tres números tecleados por usuario.
- (1.10.4) Pide al usuario una cantidad de "millas náuticas" y muestra la equivalencia en metros, usando: 1 milla náutica = 1852 metros.

1.11. 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;
```

```

class Ejemplo_01_11a
{
    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;

class Ejemplo_01_11b
{
    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:

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

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

1.12. Escribir sin avanzar de línea

En el apartado 1.7.3 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;

class Ejemplo_01_12a
{
    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;

class Ejemplo_01_12b
{
    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:

- **(1.12.1)** 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.
- **(1.12.2)** Pide al usuario un número y muestra 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

- **(1.12.3)** Crea 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".
- **(1.12.4)** 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 siguientes tablas de conversión: kelvin = celsius + 273 ; fahrenheit = celsius * 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 o no se cumplen ciertas condiciones. 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;

class Ejemplo_02_01_01a
{
    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.

A no ser que la orden "if" sea extremadamente corta, se suele partir en dos líneas para que resulte más legible, y en ese caso (frecuente), la "sentencia" se tabula un poco más a la derecha que el "if":

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

Un poco más adelante hablaremos del tamaño recomendado para esas tabulaciones.

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 en el ejercicio 2.1.1.1, habrá que ver si el resto de la división es cero: `a % b == 0`).

2.1.2. if y sentencias compuestas

Nuestro primer ejemplo de la orden "if" ejecutaba una única sentencia cuando se cumplía la condición. En un programa real, más complejo, es muy habitual que haya que dar varios pasos, no sólo uno. La forma de hacerlo es agrupar todos esos pasos entre llaves (`{` y `}`), formando lo que se conoce como una "sentencia compuesta":

```
// Ejemplo_02_01_02a.cs
// Condiciones con if (2): Sentencias compuestas
// Introducción a C#, por Nacho Cabanes
```

```

using System;

class Ejemplo_02_01_02a
{
    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 "Ejemplo_02_01_02a
}

```

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 reales" 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 tabulado un poco más a la derecha que la cabecera "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 puntuales usaremos sólo dos espacios, para que los fuentes más complejos quepan entre los márgenes del papel.

Las recomendaciones habituales para esos tamaños de tabulación son:

- No emplear el carácter de tabulación, sino **espacios** en blanco. Eso no significa que sea necesario pulsar varias veces la barra espaciadora, sino que se deberá configurar el editor que se está empleado (si, como es habitual, lo permite), para que cada pulsación de la tecla de tabulación escriba varios espacios.
- La cantidad de espacios recomendada es de **4**. En general se considera que 8 espacios es un espacio demasiado grande, y que hace que se alcance con

demasiada facilidad el límite recomendado de **80 caracteres por línea**, mientras que 2 espacios hace que sea menos legible el fuente, al distinguirse con menos facilidad el principio y final de cada bloque.

En el "mundo real", es habitual **incluir siempre** las llaves después de un "if", como medida de seguridad, porque un **fallo frecuente** es escribir una única sentencia tras "if", sin llaves, luego añadir una segunda sentencia y olvidar las llaves... de modo que la segunda orden no se ejecutará si se cumple la condición, sino siempre, como en este ejemplo:

```
// Ejemplo_02_01_02b.cs
// Condiciones con if (2b): Sentencias compuestas incorrectas
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_02b
{
    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("También puede usar negativos."); // Error
    }
}
```

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)

Los símbolos se deben escribir exactamente como aparecen en esa tabla. No se puede cambiar el orden de los que están formados por dos caracteres: es válido != pero no lo es !=, y, del mismo modo, es válido <= pero no =<.

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;

class Ejemplo_02_01_03a
{
    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.");
    }
}
```

Y otro que probara todas las condiciones sería así:

```
// Ejemplo_02_01_03b.cs
// Condiciones con if (3b): operadores relacionales
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_03b
{
    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 negativo.");
        if (numero == 0)
            Console.WriteLine("El número es cero.");
        if (numero != 0)
            Console.WriteLine("El número no es cero.");
        if (numero >= 0)
            Console.WriteLine("El número es positivo o cero.");
        if (numero <= 0)
            Console.WriteLine("El número es negativo o 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 teclea 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 el primero entre 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;

class Ejemplo_02_01_04a
{
    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;

class Ejemplo_02_01_04b
```

```
{
    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 ligeramente 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;

class Ejemplo_02_01_04c
{
    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

Operador Significado

&&	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 positivos, podría ser:

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

using System;

class Ejemplo_02_01_05a
{
    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.");
    }
}
```

Es frecuente analizar el comportamiento de estos conectores lógicos usando "**tablas de verdad**", que detallan cómo se van a comportar según si cada uno de

los valores comparados es verdadero o es falso. Por ejemplo, si unimos condiciones con "**y**", el resultado será "verdadero" sólo en el caso de que ambas lo sean (por ejemplo, "es par y positivo" sólo será verdad si el número es par y además es positivo):

<i>a</i>	<i>b</i>	<i>a && b</i>
falso	falso	falso
falso	verdadero	falso
verdadero	falso	falso
verdadero	verdadero	verdadero

De forma similar, si enlazamos dos condiciones con "**o**", el resultado será "verdadero" en cuanto una de ellas lo sea (por ejemplo, "si x es par o positivo" se cumplirá si x es par, pero también si es positivo, o si ocurren ambas cosas a la vez):

<i>a</i>	<i>b</i>	<i>a b</i>
falso	falso	falso
falso	verdadero	verdadero
verdadero	falso	verdadero
verdadero	verdadero	verdadero

Y la tabla de verdad del conector "**no**" es la más sencilla de todas: lo contrario de algo verdadero será algo falso, y viceversa (por ejemplo, "si x no es par" se cumplirá cuando no ocurra que x sea par):

<i>a</i>	<i>! a</i>
falso	verdadero
verdadero	falso

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". La mayoría de compiladores modernos al menos nos avisarán de que quizás estemos asignando un valor sin pretenderlo, pero no se trata de un "error" que invalide la compilación, sino de 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 "booleano"), de modo que obtendríamos un error de compilación "Cannot implicitly convert type 'int' to 'bool'" (*no se puede convertir implícitamente el tipo "int" en "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;

class Ejemplo_02_01_06a
{
    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, hay autores que recomiendan 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) ...
```

Aun así, la mayoría de entornos de desarrollo modernos en C y C++ detectan las asignaciones dentro de un "if" y muestran un mensaje de **aviso** (pero, insisto, no un "error fatal": la compilación continúa y se crea un ejecutable; por eso, en estos lenguajes es vital ser muy cuidadoso y prestar atención a los mensajes de aviso, no sólo a los errores).

Ejercicios propuestos:

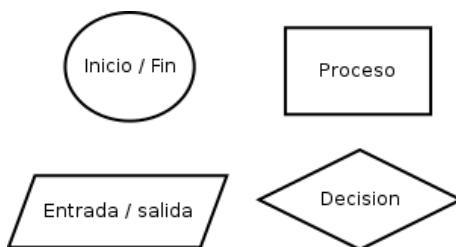
(2.1.6.1) Crea una variante del ejemplo 02_01_06a, en la que la comparación de igualdad sea correcta y en la que las variables aparezcan en el lado derecho de la comparación y los números en el lado izquierdo.

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.

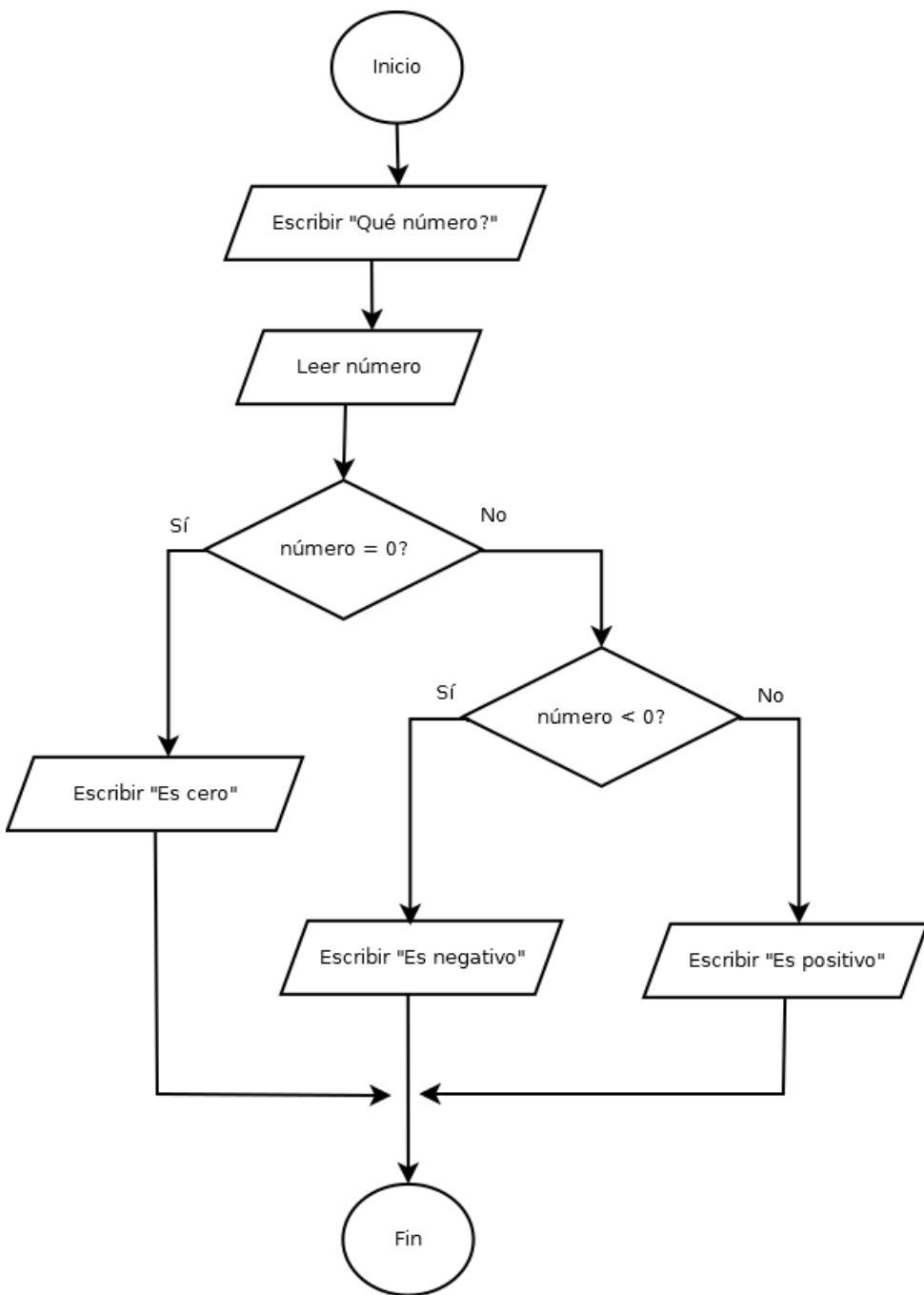
En primer lugar, vamos a ver los 4 elementos básicos de un diagrama de flujo, y luego los aplicaremos a un caso concreto.



Es decir:

- El inicio o el final del programa se indica dentro de un círculo.
- Los procesos internos, como realizar operaciones aritméticas (sumas, restas, multiplicaciones, etc.), 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 inclinados 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 cómo leer de teclado, como escribir en pantalla, y las decisiones serán un "if", que si se cumple ejecutará la sentencia que aparece en su salida "si" y si no se cumple ("else") ejecutará lo que aparezca en su salida "no".

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. Hoy en día se usa para

especificar criterios de diseño a alto nivel, al igual que el pseudocódigo, pero no como representación directa de las órdenes de un programa.

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.
- (2.1.7.4) 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, usando 3 "if" encadenados.

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. Es un formato más compacto pero también más difícil de leer. Se trata del "**operador condicional**" ?: (también conocido como "**operador ternario**"), que se emplea así:

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

y equivale a decir "si se cumple la condición, la variable *nombreVariable* debe tomar el valor *valor1*; si no, tomará 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 en otros muchos problemas reales, cuando queramos optar entre **dos valores posibles para una misma 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;

class Ejemplo_02_01_08a
{
    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);
    }
}
```

Realmente, no es necesario guardar en una variable el resultado de la condición. Se podría usar directamente, por ejemplo en un "WriteLine", así:

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

using System;

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

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

        Console.WriteLine("El mayor de los números es {0}.",
            a > b ? a : b);
    }
}
```

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

```
// Ejemplo_02_01_08c.cs
// El operador condicional (3)
// Introducción a C#, por Nacho Cabanes

using System;
```

```

class Ejemplo_02_01_08c
{
    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 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 analizar **varios posibles valores** de una misma variable, puede resultar muy pesado tener que hacerlo con muchos "if" seguidos o encadenados:

```

// Ejemplo_02_01_09a.cs
// Acercamiento a "switch"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_09a
{
    static void Main()
    {
        int n;

        Console.Write("Introduce un número del 1 al 5: ");
        n = Convert.ToInt32(Console.ReadLine());

        if (n==1)
            Console.WriteLine("Uno");
        else if (n==2)
            Console.WriteLine("Dos");
        else if (n==3)
            Console.WriteLine("Tres");
    }
}

```

```

        else if (n==4)
            Console.WriteLine("Cuatro");
        else if (n==5)
            Console.WriteLine("Cinco");
        else
            Console.WriteLine("No es del uno al cinco");
    }
}

```

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 (habitualmente será el nombre de una variable).
- Después, tras varias órdenes **case** se indica cada uno de los valores posibles, seguidos por un símbolo de "dos puntos".
- A continuación de cada "case" se indican los pasos (porque pueden ser varios) que se deben dar si la expresión tiene ese valor concreto, terminando con **break**.
- Si es necesario hacer algo en caso de que no se cumpla ninguna de las condiciones (por ejemplo, responder con un mensaje de error), 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.

Así, una versión alternativa del programa anterior podría ser:

```

// Ejemplo_02_01_09b.cs
// Contacto con "switch"
// Introducción a C#, por Nacho Cabanes

```

```

using System;

class Ejemplo_02_01_09b
{
    static void Main()
    {
        int n;

        Console.WriteLine("Introduce un número del 1 al 5: ");
        n = Convert.ToInt32(Console.ReadLine());

        switch(n)
        {
            case 1: Console.WriteLine("Uno"); break;
            case 2: Console.WriteLine("Dos"); break;
            case 3: Console.WriteLine("Tres"); break;
            case 4: Console.WriteLine("Cuatro"); break;
            case 5: Console.WriteLine("Cinco"); break;
            default: Console.WriteLine("No es del uno al cinco"); break;
        }
    }
}

```

Vamos a ver otro 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 saber 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_09c.cs
// La orden "switch" y caracteres
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_09c
{
    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("Digito.");
                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_09d.cs
// La orden "switch" (variante sin break)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_09d
{
    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("Número");
                        break;
            default:   Console.WriteLine("Ni espacio");
                        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_09e.cs
// La orden "switch" con cadenas de texto
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_01_09e
{
    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("Hola, desconocido.");
                break;
        }
    }
}
```

Ejercicios propuestos:

(2.1.9.1) Crea un programa que pida un número del 1 al 10 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: necesitarás 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".
- (2.1.9.7)** Pide al usuario el número de un día de la semana y escribe su nombre (por ejemplo, si escribe 2, la respuesta debería ser "Martes"). Hazlo primero con "if" y después con "switch".
- (2.1.9.8)** Pide al usuario el un número de mes y escribe su nombre (por ejemplo, si escribe 3, la respuesta debería ser "Marzo"), usando "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**"). Tenemos varias formas de conseguirlo, según si queremos comprobar la condición antes de repetir, o después de cada repetición, o realizar la tarea una cierta cantidad de veces.

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 }. Como ocurría con "if", puede ser **recomendable** incluir siempre las llaves, aunque se trate de una única sentencia, para evitar errores posteriores difíciles de localizar.

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

```
// Ejemplo_02_02_01a.cs
// La orden "while": mientras...
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_01a
{
    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 volvérse 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. Se repetirá mientras los dos números sean distintos de cero (terminará cuando uno de ellos sea cero).
- (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;

class Ejemplo_02_02_01_02a
{

```

```

static void Main()
{
    int n = 1;

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

```

Y esta misma estructura se puede emplear también para hacer algo varias veces, aunque no se muestre el valor del contador en pantalla, como en este ejemplo, que escribe 5 letras X:

```

// Ejemplo_02_02_01_02b.cs
// Contar con "while" (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_01_02b
{
    static void Main()
    {
        int n = 1;

        while (n < 6)
        {
            Console.Write("X");
            n = n + 1;
        }
        Console.WriteLine(); // Para avanzar de línea al final
    }
}

```

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)** Haz un programa que muestre tantos asteriscos (en la misma línea) como indique el usuario.

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;

class Ejemplo_02_02_02a
{
    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;

class Ejemplo_02_02_02b
{
    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)** Crea 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;
```

Es muy habitual usar la letra "i" (abreviatura de "índice") como contador cuando se trata de tareas muy sencillas. Así, para **contar del 1 al 10**, tendríamos "i=1" como valor inicial, "i<=10" como condición de repetición, y el incremento sería de 1 en 1, con "i=i+1".

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

La orden para **incrementar** en una unidad 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, en fragmentos de programa que no sean triviales, 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;

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

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

Ejercicios propuestos:

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

(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).

(2.2.3.3) 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 abbreviar $i--$).

Nota: como veremos también con más detalle en el próximo tema, existe una notación abreviada para incrementar en varias unidades el valor de una variable: " $i = i+5$ " se puede escribir de forma alternativa como " $i += 5$ " (no debe existir separación entre el símbolo "+" y el símbolo "="), y también se puede hacer lo mismo para decrementar en varias unidades: " $i = i-4$ " se puede escribir como " $i -= 4$ ". Así, se puede mostrar los números pares del 2 al 10 con

```
for (i=2; i<=10; i+=2)
    ...

```

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 jamás 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 evidente de fragmento de programa 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" o 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, separados por un espacio en blanco.

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;

class Ejemplo_02_02_05a
{
    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, usando dos bucles anidados.

2.2.6. Repetir sentencias compuestas

En los últimos ejemplos que hemos visto, después de "for" había una única sentencia. Al igual que ocurría con "if" y con "while", 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;

class Ejemplo_02_02_06a
{
    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();
        }
    }
}
```

Si repetimos la escritura de varias líneas, cada una formadas por varias columnas, podremos dibujar varias figuras geométricas sencillas (bastantes de las cuales quedarán propuestas para que tú las intentes). Por ejemplo, se puede dibujar un rectángulo con:

```
// Ejemplo_02_02_06b.cs
// Rectángulo de asteriscos
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_06b
{
    static void Main()
    {
        int fila, columna;
        int alto = 5;
        int ancho = 10;

        for (fila=1; fila <= alto; fila++)
        {
```

```
        for (columna=1; columna <= ancho; columna++)
            Console.Write("*");
        Console.WriteLine();
    }
}
```

que tendría como resultado:

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:

(2.2.6.3) Haz un programa que dibuje un cuadrado de asteriscos, cuyo ancho (y alto, que tendrá el mismo valor) será introducido por el usuario.

(2.2.6.4) Crea un triángulo de asteriscos, que mostrará uno en la primera fila, dos en la segunda, tres en la tercera y así sucesivamente, hasta llegar al tamaño indicado por el usuario.

(2.2.6.5) Dibuja un triángulo de asteriscos descendente. Por ejemplo, si el usuario escoge "4" como tamaño, la primera fila tendrá 4 asteriscos, la segunda tendrá 3, la siguiente tendrá 2 y la última tendrá 1.

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" (pronto hablaremos más de ese tipo de datos) 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;

```
class Ejemplo_02_02_07a  
{  
    static void Main()
```

```

{
    char letra;

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

```

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

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 cambia su valor. 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;

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

        for (letra='z'; letra>='a'; letra--)
            Console.WriteLine("{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, empleando dos "for" anidados.

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
// Reutilización incorrecta de la variable de un "for"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_08a
{
    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 reutilización incorrecta de la variable
// de un "for": no compila
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_08b
{
    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 se 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;

class Ejemplo_02_02_08c
{
    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

Como ya hemos comentado, las "llaves" no son necesarias cuando una orden "for", "while", "do-while" o "if" va a repetir una única sentencia, sino sólo cuando se repite un bloque de dos o más sentencias. Pero un error frecuente consiste repetir inicialmente una única orden, añadir después una segunda orden repetitiva y olvidar las llaves. Otro error (menos frecuente) es querer incluir varias órdenes dentro de una de estas estructuras, tabular estas nuevas órdenes más a la derecha pero olvidar las llaves. Por eso, en programas no triviales es muy recomendable incluir siempre las llaves, aunque esperemos repetir sólo 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;

class Ejemplo_02_02_09a
{
    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();
    }
}
```

Si incluimos llaves, incluso donde no son imprescindibles, el problema desaparece:

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

using System;

class Ejemplo_02_02_09b
{
    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 y 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;

class Ejemplo_02_02_10a
{
    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 comprueba sólo si cumple la condición de repetición del bucle, sino que además éste se puede interrumpir por otros criterios. Como norma general, es preferible reescribir la condición del bucle de otra forma. En el ejemplo anterior, bastaría que la condición de "if" fuese "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;

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

            Console.WriteLine("{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;

class Ejemplo_02_02_10c
{
    static void Main()
    {
        int contador=105;

        while ( (contador<=120) && (contador % 13 != 0) )
        {
            Console.WriteLine("{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, interrumpe la búsqueda con "break").

(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 con "break").

(2.2.10.3) Crea una versión alternativa del ejercicio 2.2.10.1 (máximo común divisor) usando "while", en vez de "for" y "break".

(2.2.10.4) Crea una versión alternativa del ejercicio 2.2.10.2 (mínimo común múltiplo) usando "while", en vez de "for" y "break".

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;

class Ejemplo_02_02_11a
{
    static void Main()
    {
        for (int contador=1; contador<=10; contador++)
        {
            if (contador==5)
                continue;

            Console.WriteLine("{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. Al igual que ocurre con "break", su uso está desaconsejado. Como alternativa más legible, se podría haber utilizado un "if" opuesto al anterior, 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;

class Ejemplo_02_02_11b
{
    static void Main()
    {
```

```

        for (int contador=1; contador<=10; contador++)
    {
        if (contador != 5)
            Console.WriteLine("{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".
- (2.2.11.3)** Crea una versión alternativa del ejercicio 2.2.11.1, que no utilice "continue" sino el "if" contrario.
- (2.2.11.4)** Crea una versión alternativa del ejercicio 2.2.11.2, que no emplee "continue" sino el "if" contrario.

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;

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

        Console.WriteLine();

        int n=1;
        while (n<=10)
        {
            Console.WriteLine("{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;

class Ejemplo_02_02_12b
{
    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 valor 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.WriteLine("{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.WriteLine("{0} ",i);
```

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

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

2.2.14. Saltos incondicionales: 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 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 etiqueta;
```

y la posición de salto se indica con una "etiqueta", un nombre seguido de un símbolo de dos puntos (:)

```
etiqueta:
```

como en el siguiente ejemplo, que escribe "Pasada 1", "Pasada 2" y así sucesivamente hasta que se detenga la ejecución cerrando la ventana del programa, porque el salto incondicional hacia un punto anterior crea un "bucle sin fin":

```
// Ejemplo_02_02_14a.cs
// Salto incondicional con "goto"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_02_14a
{
    static void Main()
    {
        int cantidad = 1;

        repetir:
        Console.WriteLine("Pasada {0}; ", cantidad);
        cantidad++;
        goto repetir;
    }
}
```

Un ejemplo de uso más real, para salir de un bucle muy anidado, podría ser:

```
// Ejemplo_02_02_14b.cs
// "for" y "goto"
// Introducción a C#, por Nacho Cabanes
```

```

using System;

class Ejemplo_02_02_14b
{
    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", mientras que el programa "casi equivalente" que emplea "break" no se comporta igual, sino que continúa y da todas las pasadas para $i=2$, $i=3$, $i=4$ e $i=5$.

```

// Ejemplo_02_02_14c.cs
// "break" en vez de "goto"
// Introducción a C#, por Nacho Cabanes

```

```

using System;

class Ejemplo_02_02_14c
{
    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))
            break;
        Console.WriteLine("i vale {0} y j vale {1}.", i, j);
    }

    Console.WriteLine("Fin del programa");
}

```

Ejercicios propuestos:

(2.2.14.1) 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".

2.2.15. "for" y el operador coma

Cuando hemos empleado la orden "for", siempre hemos usado una única variable como contador. Esto es, con diferencia, lo más habitual, pero no tiene por qué ocurrir siempre. Vamos a verlo con un ejemplo:

```

// Ejemplo_02_02_15a.cs
// Operador coma
// Introducción a C#, por Nacho Cabanes

using System;

class OperadorComa
{
    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
```

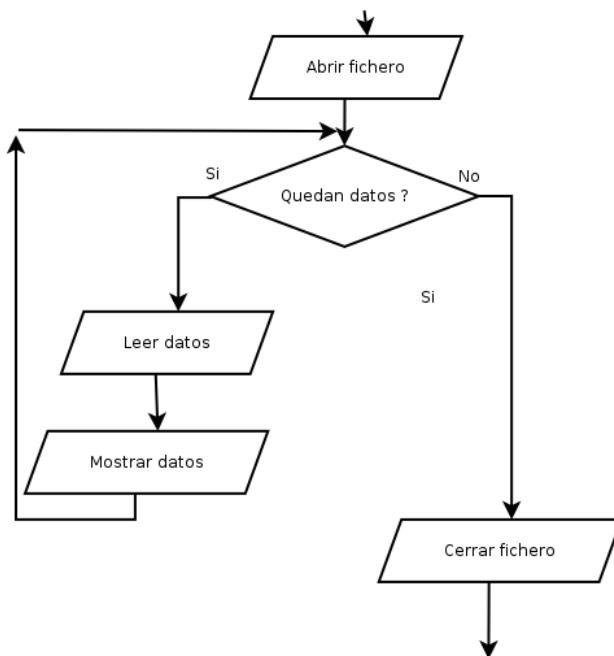
Nota: En el lenguaje C se puede "rizar el rizo" todavía un poco más: la condición de terminación también podría tener una coma, y entonces no se sale del bucle "for" hasta que se cumplen las dos condiciones (algo que no es válido en C#, ya que la condición debe ser un "booleano", algo que tenga como resultado "verdadero" o "falso", y la coma no es un operador válido para operaciones booleanas):

```
for (i=0, j=1; i<=5, j<=30; i++, j+=2)
```

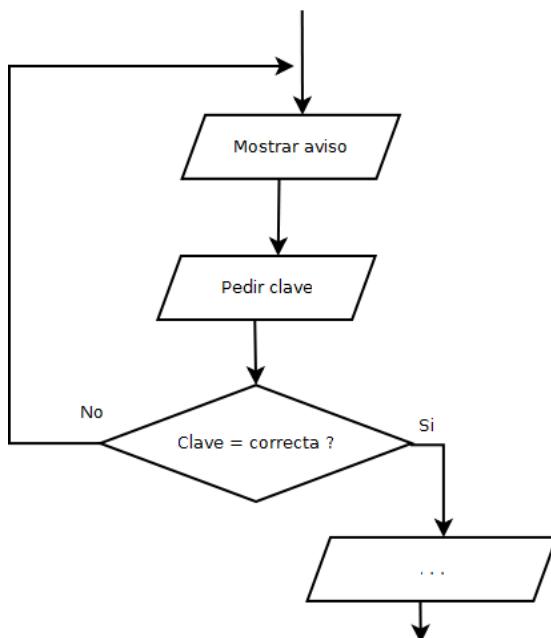
2.3. Más sobre diagramas de flujo. Diagramas de Nassi-Shneiderman

En el apartado 2.1.7 tuvimos un contacto con la forma de ayudarnos de los diagramas de flujo para plantear lo que un programa debe hacer. Si entendemos esta herramienta, el paso de esta notación 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):



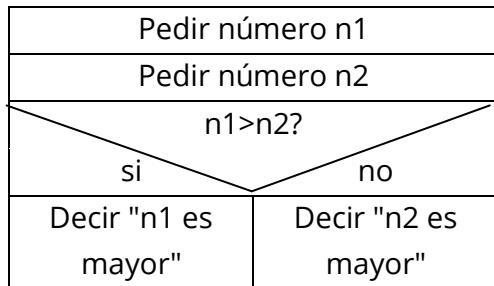
Y un "**do...while**" se representaría como una condición al final de un bloque que se repite:



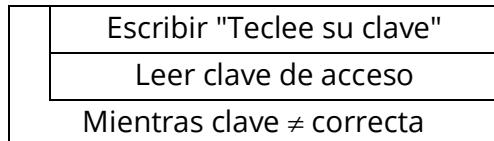
Aun así, existen otras notaciones más modernas y que pueden resultar más cómodas. Sólo comentaremos una: los **diagramas de Nassi-Shneiderman**, diagramas de cajas o diagramas de Chapin. En ellos se representa cada orden dentro de una caja, y el conjunto del programa es una serie de cajas apiladas de arriba (primera orden) a abajo (última orden):

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

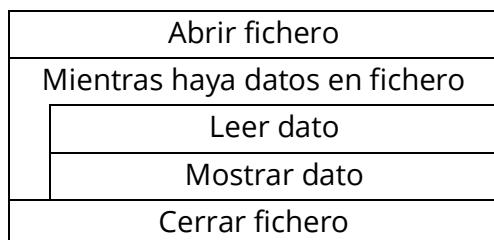
Las **condiciones** se denotan dividiendo las cajas en dos:



Y las condiciones repetitivas se indican dejando una barra vertical 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**):



Los "**for**" se suelen mostrar como un "while", pero detallando los valores: "para n = 1 hasta 5".

En la práctica, ambas notaciones **se usan poco** a nivel de programación formal, porque un programa simple puede necesitar más tiempo para representarse con una notación gráfica como éstas que para ser tecleado, y la diferencia es aún mayor cuando hay que hacer alguna modificación, que son más costosas en una notación gráfica que en un programa convencional. Aun así, en un momento inicial del aprendizaje (como éste), pueden ayudar a asentar conceptos básicos, como

decidir qué debe abarcar un "if" o si se debe optar por un "while" o un "do-while". En "el mundo real", se usan más en **diseño de alto nivel de programas complejos** que como paso previo en programas sencillos.

Ejercicios propuestos:

- (2.3.1) Crea el diagrama de Nassi-Shneiderman para el programa que pide dos números al usuario y dice cuál es el mayor de los dos. Compáralo con el diagrama de flujo del ejercicio 2.1.7.1.
- (2.3.2) Crea el diagrama de Nassi-Shneiderman 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. Compáralo con el diagrama de flujo del ejercicio 2.1.7.2.
- (2.3.3) Crea el diagrama de Nassi-Shneiderman para el programa que pide tres números al usuario y dice cuál es el mayor de los tres. Compáralo con el diagrama de flujo del ejercicio 2.1.7.3.
- (2.3.4) Crea el diagrama de flujo y el diagrama de Nassi-Shneiderman para un programa que pida pares de números al usuario y muestre el resultado dividir el primero entre el segundo, repitiendo hasta que el segundo número sea cero.

2.4. *foreach*

Nos queda por ver otra orden que permite hacer cosas repetitivas: "foreach" (se traduciría "para cada"). La veremos con detalle 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_04a.cs
// Primer ejemplo de "foreach"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_04a
{
    static void Main()
    {
        Console.Write("Dime tu nombre: ");
        string nombre = Console.ReadLine();
        foreach(char letra in nombre)
        {
            Console.WriteLine(letra);
        }
    }
}
```

Ejercicios propuestos:

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

2.5. 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 o no existe, no habrá datos que leer).

De igual modo, "**do...while**" será lo adecuado cuando debamos repetir al menos una vez (por ejemplo, para pedir una clave de acceso, se le debe preguntar al menos una vez al usuario, o quizás más veces, si no 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 podría ser "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 (utiliza en cada caso el tipo de bucle que consideres más adecuado):

(2.5.1) Crea un programa que muestre una cuenta atrás (3 2 1 0) desde el número que introduzca el usuario hasta cero. Ese número debe estar entre 1 y 10 (y el programa debe comprobar que realmente lo está, y volverlo a pedir tantas veces como sea necesario, en caso de que no sea así).

(2.5.2) Crea un programa en el que el usuario deba adivinar un número del 1 al 100 (prefijado en el programa). En cada pasada deberá avisar de si se ha pasado o se ha quedado corto.

(2.5.3) Haz 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.

(2.5.4) Crea un programa que pida un número al usuario y diga si es primo (divisible sólo entre 1 y él mismo).

(2.5.5) Crea un programa que descomponga un número (que teclee el usuario) como producto de su factores primos. Por ejemplo, $60 = 2 \cdot 2 \cdot 3 \cdot 5$ (pista: como primera aproximación, puedes escribir siempre un "punto" después de cada

número y luego terminar con la cifra uno, así: $60 = 2 \cdot 2 \cdot 3 \cdot 5 \cdot 1$; cuando lo consigas, piensa cómo harías para eliminar ese " $\cdot 1$ " del final).

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

(2.5.7) 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:

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

(2.5.8) 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:

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

(2.5.9) 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
```

(2.5.10) 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
```

2.6. Una alternativa para el control errores: las excepciones

La forma "clásica" del control de errores es utilizar 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_06a.cs
// Excepciones (1)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_06a
{
    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());
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.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_06b.cs
// Excepciones (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_02_06b
{
    static void Main()
    {
        int numero1, numero2, resultado;

        try
        {
            Console.WriteLine("Introduzca el primer numero");
            numero1 = Convert.ToInt32(Console.ReadLine());
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

```
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 arriesgada es "probar el programa" y ver qué errores obtenemos en pantalla al introducir un valor no válido. Esta alternativa es la menos deseable, porque quizás pasemos por alto algún tipo de error que pueda surgir y que nosotros no hayamos previsto. En cualquier caso, volveremos a las excepciones más adelante.

Ejercicios propuestos:

(2.6.1) 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 y se detendrá. Lo mismo ocurrirá si el año de nacimiento no es un número válido.

(2.6.2) 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.

2.7. 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 transcurre realmente nuestro programa, y también nos dejan analizar los valores de las variables en cada momento.

Veremos como ejemplo el caso de Visual Studio 2015 Community, pero las diferencias con otras versiones de este entorno deberían ser mínimas. Vamos a partir de un programa sencillo que manipule un par de variables, como:

```
// Ejemplo_02_07a.cs
// Modificación de una variable para depuración
// Introducción a C#, por Nacho Cabanes

using System;

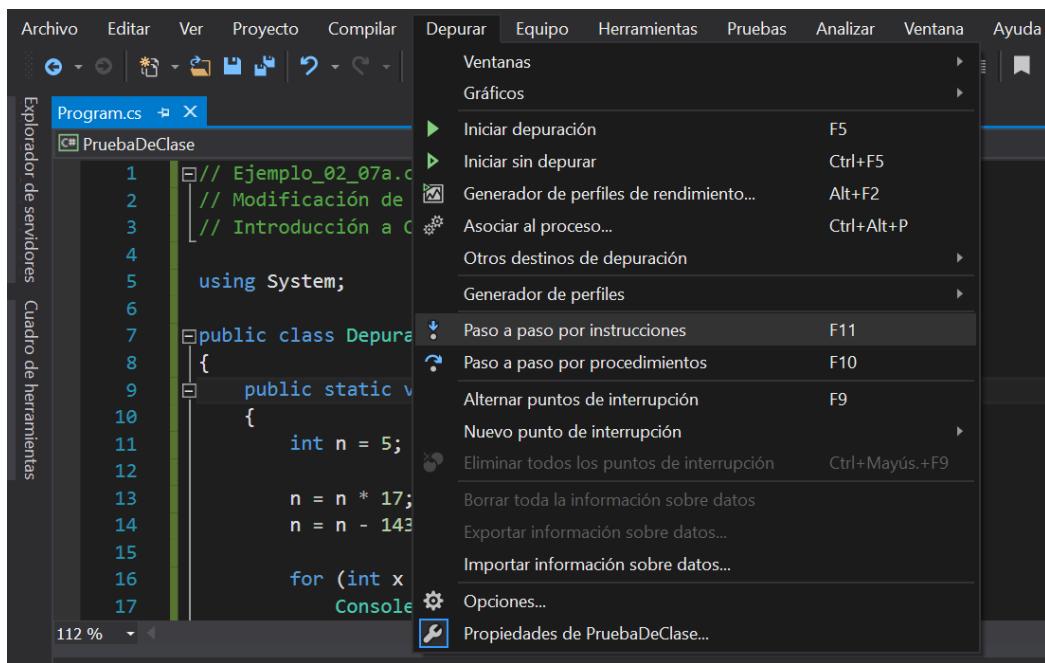
class Depuracion
{
    static void Main()
    {
        int n = 5;

        n = n * 17;
        n = n - 1432;

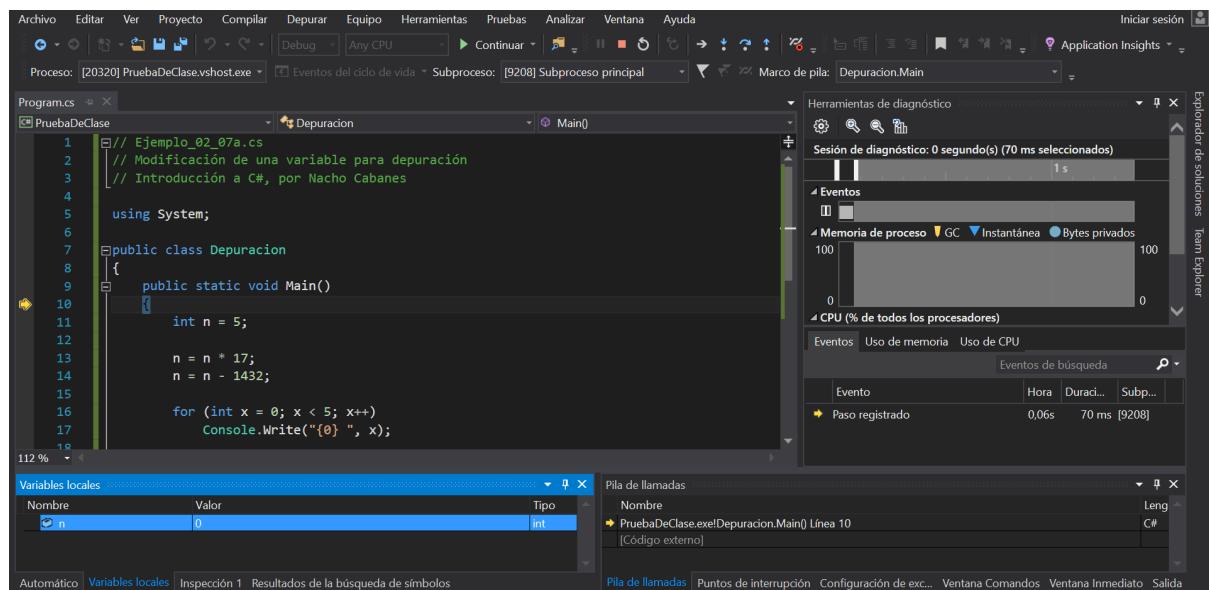
        for (int x = 0; x < 5; x++)
            Console.WriteLine("{0} ", x);

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

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" (a la que corresponde la tecla F11):



Si escogemos esa opción del menú o pulsamos F11, aparece una ventana inferior con la lista de variables (initialmente, sólo "n"), y una flecha amarilla que señala el punto del programa en el que nos encontramos (actualmente 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, realzando con fondo amarillo la siguiente línea que se analizará:

```

1 // Ejemplo_02_07a.cs
2 // Modificación de una variable para depuración
3 // Introducción a C#, por Nacho Cabanes
4
5 using System;
6
7 public class Depuración
8 {
9     public static void Main()
10    {
11        int n = 5;
12
13        n = n * 17;
14        n = n - 1432; ≤ 1 ms transcurridos
15
16        for (int x = 0; x < 5; x++)
17            Console.WriteLine("{0} ", x);
18    }
19 }

```

Variables locales

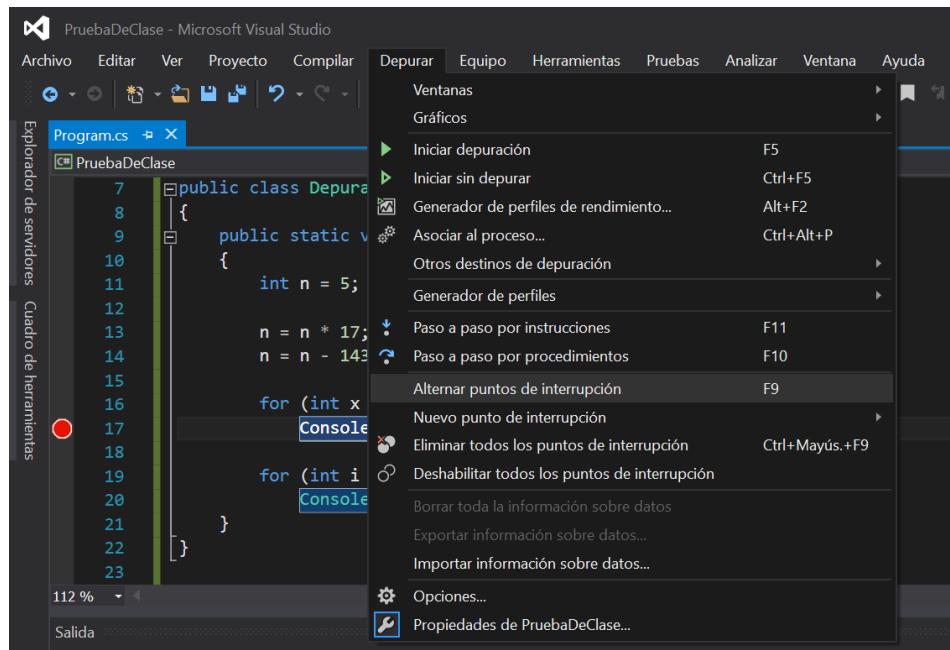
Nombre	Valor	Tipo
n	85	int

Pila de llamadas

Nombre
PruebaDeClase.exe!Depuración.Main [Código externo]

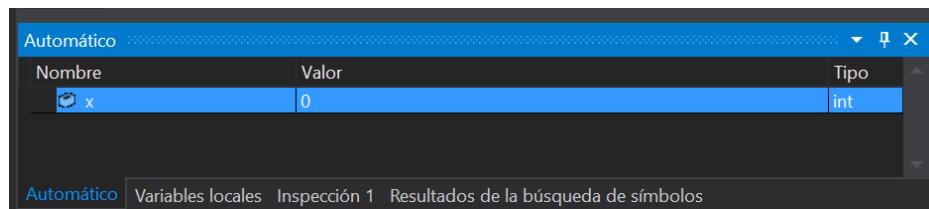
En este caso, hemos dado dos pasos: $n = 5$ y $n = n * 17$, de modo que actualmente n vale 85. El cursor está en la siguiente línea que queremos se va a procesar, que aparece destacada con fondo amarillo.

En este primer contacto, hemos avanzado paso 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 parte de programa. La forma de conseguirlo es desplazarnos hasta la primera línea que queramos analizar y escoger otra de las opciones del menú de depuración: "Alternar puntos de interrupción" (tecla F9). Aparecerá una marca de color rojo en la línea actual. Como alternativa, podemos hacer clic con el ratón en el margen izquierdo del programa, junto a esa línea:

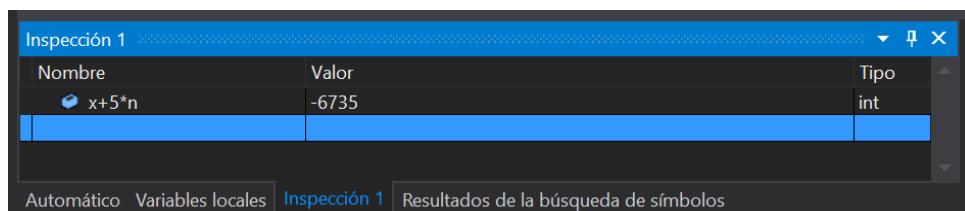


Si ahora iniciamos la depuración del programa, avanzará hasta ese punto, y será en esa línea en la que se detenga. A partir de ahí, podemos seguir depurando paso a paso como antes, pulsando F11.

Si tenemos muchas variables, nos puede interesar más la pestaña "Automático" que la de "Variables locales", porque ésta mostrará sólo aquellas que han cambiado recientemente y no veremos las que Visual Studio considere que son menos relevantes en este momento:

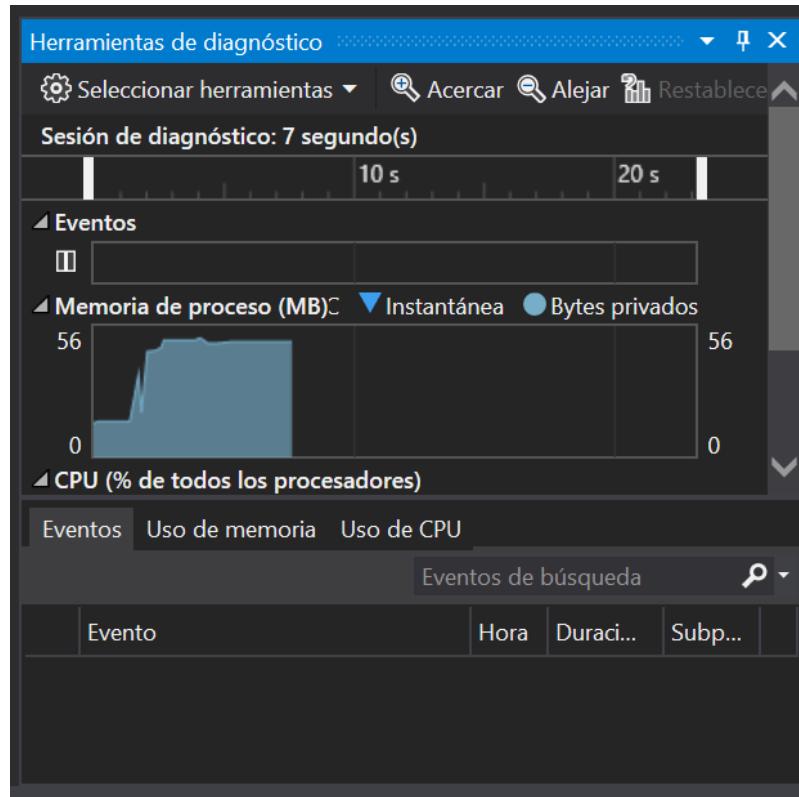


Además, existe también una pestaña "Inspección", en la que podemos escribir nosotros cualquier expresión cuyo valor queramos analizar:



Y en ciertas versiones de Visual Studio es posible que se nos muestre también en la parte derecha de la pantalla una ventana de "Herramientas de diagnóstico",

que, entre otros detalles, nos puede informar del consumo de memoria y de CPU por parte de nuestro programa. En programas más complejos, un uso de memoria que aumente continuamente sería un claro indicador de un error en nuestro programa:



3. Tipos de datos básicos

3.1. Tipo de datos entero

Hemos hablado de números enteros, de cómo realizar operaciones sencillas con valores prefijados 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 refinarn, de dar más detalles. El primer "matiz" importante que hemos esquivado hasta ahora es el "tamaño" de los números que podemos emplear en nuestros programas, así como su signo (positivo o negativo), y esos son detalles importantes cuando se emplean muchos de los lenguajes que derivan de C (como C#, C++ y Java). 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. Por ello, los "int" no serán adecuados si necesitamos almacenar números de más de 10 cifras, ni si debemos emplear cifras decimales.

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

Un "int" no es la única alternativa disponible. Por ejemplo, si deseamos guardar la edad de una persona, no necesitamos usar números negativos, y nos bastaría con 3 cifras, así que es de suponer que existiría 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 además usar cifras decimales, pero eso lo dejaremos para el siguiente apartado, el 3.2.

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

(Detalle para alumnos **avanzados**: la "u" que precede el nombre de algunos tipos de datos indica que sólo permiten valores positivos, como en "uint", abreviatura de "unsigned int", entero sin signo; por el contrario, la "s" de "sbyte" se refiere a que es un "signed byte", un byte con signo. En los tipos de datos que permiten guardar datos positivos y negativos, el primer "bit" se emplea para el signo, por lo que los valores que se pueden almacenar son "más pequeños", al disponer de un bit menos de información).

Como se puede observar en la tabla anterior, el tipo de dato más razonable para guardar edades sería "byte", que permite valores entre 0 y 255, y ocupa la cuarta parte que un "int".

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

using System;

class Ejemplo_03_01_01a
{
    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 pedir al usuario datos de esos otros tipos de datos enteros, ya no nos servirá Convert.ToInt32, porque ya no se tratará de enteros de 32 bits (4 bytes).

Así, para datos de tipo "byte" usaremos Convert.ToByte (sin signo) y ToSByte (con signo), para datos de 2 bytes (short) tenemosToInt16 (con signo) y ToUInt16 (sin signo), y para los de 8 bytes (long) existenToInt64 (con signo) y ToUInt64 (sin signo). De igual modo, para los enteros de 32 bits sin signo se empleará ToUInt32.

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

using System;

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

        byte edad = Convert.ToByte(ejemplo1);
        ushort anyo = Convert.ToInt16(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 "int", 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;

Un ejemplo básico de su uso es:

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

using System;

class Ejemplo_03_01_03a
{
    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);
    }
}
```

En general, cuando queramos incrementar o decrementar sólo en una unidad el valor de una variable, será preferible usar la notación "x++" en vez de "x=x+1", porque eso ayudará al compilador a generar un código máquina más eficiente.

La operación incremento tiene algo más de dificultad de la que puede parecer en un primer vistazo: en C# (y los lenguajes que derivan de C) es posible dar un valor a una variable a la vez que se incrementa otra: $y = x++$;

En asignaciones como esas, se puede distinguir entre "preincremento" y "postincremento". Por ejemplo, en la operación

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

Si "a" valía 2, lo que esta instrucción hace es dar a "b" el valor de "a" y después 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 aumenta "a" y luego se asigna ese valor a "b" (**preincremento**), de modo que a=3 y b=3.

Un ejemplo más detallado:

```
// Ejemplo_03_01_03b.cs
// Preincremento y postincremento
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_01_03b
{
    static void Main()
    {
        int m = 10;
        int n = 10;

        int a = m++;
        Console.WriteLine("a vale {0}", a);
        Console.WriteLine("m vale {0}", m);

        int b = ++n;
        Console.WriteLine("b vale {0}", b);
        Console.WriteLine("n vale {0}", n);
    }
}
```

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

Ejercicios propuestos:

(3.1.3.1) Crea un programa que use tres variables enteras 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; Calculalo 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 operaciones como "a = a+5". Estas son las abreviaturas más habituales:

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;

class Ejemplo_03_01_04a
{
    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);
    }
}
```

Al igual que ocurría con el incremento y decremento en una unidad, estas operaciones permiten al compilador generar un código máquina más eficiente que para la operación equivalente no abreviada.

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;

class Ejemplo_03_01_05a
{
    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);
    }
}
```

3.1.6. Operaciones con bits

C# nos permite realizar operaciones con los bits de uno o dos números (AND, OR, XOR, etc). Vamos primero a ver qué significa cada una de esas operaciones, para después aplicarlo a un ejemplo completo:

Operación	Resultado	En C#	Ejemplo
Complemento (not)	Cambiar 0 por 1 y viceversa	~	$\sim 1100 = 0011$
Producto lógico (and)	1 sólo si los 2 bits son 1	&	$1101 \& 1011 = 1001$
Suma lógica (or)	1 sólo si uno de los bits es 1		$1101 1011 = 1111$
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	>>	$1101 >> 2 = 0011$

Un ejemplo de su uso podría ser:

```

// Ejemplo_03_01_06a.cs
// Operaciones a nivel de bits
// Introducción a C#, por Nacho Cabanes

using System;

class Bits
{
    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: $1 \wedge 1 = 0$, $1 \wedge 0 = 1$, $0 \wedge 0 = 0$

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 equivale a 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 equivale a dividir por potencias de dos; la suma lógica exclusiva (xor) es un método rápido, sencillo y reversible 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

(3.1.6.1) Crea un programa que pida al numero del 0 al 255 y muestre el resultado de hacer un XOR con un cierto dato prefijado (y también en ese rango). Comprueba que la operación es reversible (por ejemplo, $131 \text{ xor } 5 = 134$, y $134 \text{ xor } 5 = 131$).

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 ocurría con los números enteros, tendremos más de un tipo de número real para elegir.

3.2.1. Coma fija y coma flotante

Existen dos formas habituales de almacenar números reales dentro de un sistema informático:

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 de forma correcta (tiene más de 3 cifras enteras). Esta forma de almacenar números reales no se suele emplear en los sistemas informáticos modernos.

Coma flotante: la cantidad de decimales y de cifras enteras permitida es variable, lo que importa es la cantidad 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 permite emplear números de coma flotante. En C# corresponden al tipo de datos llamado "float" (aunque hay más posibilidades, como veremos dentro de poco).

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

using System;

class Ejemplo_03_02_01a
{
    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 correctamente 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

```

Si algún numero real prefijado contiene decimales, tendremos que añadirle el **sufijo "f"**, como en este ejemplo:

```
float f1 = 2.5f, f2 = 3.06f;
```

Un detalle importante que quizá hayas pasado por alto: en la secuencia de órdenes `int i1 = 2, i2 = 3; float divisionI; divisionI = i1/i2;` la variable que guarda el resultado es "float", pero la operación se realiza entre dos números enteros, luego su resultado es un número entero, con valor 0, y ese valor es el que se almacena en la variable "float"; en el segundo caso, la operación se realiza entre números reales, luego su resultado es un número real desde el primer momento.

Otro detalle importante, consecuencia de la **precisión limitada** de los números reales, es que será peligroso comparar igualdad de valores entre dos expresiones. Por ejemplo, puede ocurrir que esperemos que el resultado de una operación sea 1 y que realmente obtengamos 0.999999 o bien 1.000001. Por eso, será preferible no hacer comparaciones con números reales como "if ($x==1$)" sino mirar si el valor está dentro de un determinado rango, como en "if (($x > 0.9999$) && ($x < 1.0001$))".

Ejercicios propuestos:

~~(3.2.1.1) Crea un programa que muestre el resultado de dividir 3 entre 4, primero 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") disponemos del 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

(**Nota:** los tipos "float" y "double" son frecuentes en muchos lenguajes de programación, pero el tipo "decimal" es menos habitual).

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;

class Ejemplo_03_02_02a
{
    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 0,6666666666666667
```

En el caso de los números de doble precisión, no será necesario ningún sufijo "f" (precisamente porque esa "f" sirve para indicar al compilador que ese dato deberá almacenarse en el espacio correspondiente a un "float", aunque eso suponga perder precisión):

```
double f1 = 2.5, f2 = 3.06;
```

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_02b.cs
// Números reales: valor inicial de un float y de un double
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_02_02b
{
    static void Main()
    {
        int radio;
        float piFloat = 3.141592653589793238f; // Atención a la "f" del final
        double piDouble = 3.141592653589793238; // Sin "f"

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

Y su resultado sería algo como

```
Introduce el radio
5
La longitud de la circunferencia (float) es
31,41593
La longitud de la circunferencia (double) es
31,4159265358979
```

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 ($\text{área} = \pi * \text{radio}^2$). Usa datos de doble precisión.

3.2.3. Pedir números reales al usuario

Si necesitamos que sea el usuario quien introduzca los datos, deberemos leerlos como cadena de texto, y convertir al tipo adecuado cuando vayamos a realizar operaciones aritméticas, al igual que hacíamos con los enteros. Ahora usaremos `Convert.ToDouble` cuando se trate de un dato de doble precisión ("double"), `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;

class Ejemplo_03_02_03a
{
    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: en el fuente debemos escribir los decimales usando **un punto**, como 123.456, porque el lenguaje C# se apoya en construcciones del idioma inglés. Pero 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 10, donde 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 * radio al cubo * 4/3)

(3.2.3.2) Crea un programa que pida al usuario 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 = $E(1 + i)^n$. Aplicalo 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 ". Te será fácil si dibujas la gráfica "girada", de forma que los valores de " y " crezcan hacia la derecha, así:

(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 (habitual entre dos datos numéricos, no tanto para convertir de número a

texto y viceversa). 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;

class Ejemplo_03_02_04a
{
    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". (Recuerda: como viste al hacer el ejercicio 3.2.3.3, la raíz cuadrada de un número x se calcula con Math.Sqrt(x)).

(3.2.4.2) Crea una nueva versión del 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 operaciones con "double", pero mostrar el resultado como "float".

3.2.5. Formatear números

En más de una ocasión nos interesaría 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 bien 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 cifra en esa posición.
- Una almohadilla (#) indica una posición en la que puede aparecer un número, y no se escribirá nada si no existe una cifra en esa posición.
- 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;

class Ejemplo_03_02_05a
{
    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 datos de simple precisión. 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 = hipotenusa, 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 = $4 * \pi * \text{radio}^2$; volumen = $4/3 * \pi * \text{radio}^3$). 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;

class Ejemplo_03_02_06a
{
    static void Main()
    {
        int numero = 247;

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

Su resultado sería:

```
f7
11110111
```

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

Como curiosidad, para convertir números a sistema hexadecimal también se puede usar `numero.ToString("x")` si se desea obtener la cifra hexadecimal en minúsculas, o bien `numero.ToString("X")` para obtenerla en mayúsculas. No existe un formato (tan) abreviado para convertir a sistema binario.

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 = 0x23;" (donde n1 tendría el valor $16^2 + 3 \cdot 1 = 35$, expresado en base 10). En los lenguajes C y C++, un valor precedido por "0" se considera **octal**, de modo que para "int n2 = 023;" el valor decimal de n2 sería $8^2 + 3 \cdot 1 = 17$, pero en C# no es así: un número que empieza 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;

class Ejemplo_03_02_06b
{
    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

En la **versión 7** de la especificación del lenguaje C# (utilizable con **Visual Studio 2017** y posteriores, pero quizás no disponible en otros casos, por ejemplo si usas Geany y el compilador de línea de comandos) se añade también la posibilidad de especificar números en binario con el prefijo 0b:

```
int n6 = 0b10011101;
```

y de usar una "barra baja" (símbolo de subrayado) como separador en los millares (o, en realidad, en cualquier posición):

```
int cincoMillones = 5_000_000;
```

o incluso de combinar ambas posibilidades:

```
int n6 = 0b1001_1101;
```

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".

3.2.7. Funciones matemáticas

En C# disponemos de 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

Todas ellas se usan **precedidas por "Math."**

Casi todas ellas reciben datos de tipo "double". En el caso de las funciones trigonométricas, el ángulo se debe indicar en radianes, no en grados, así que en ocasiones será necesario convertir de una unidad a otra, teniendo en cuenta que la equivalencia es: 180 grados = PI radianes.

También tenemos una serie de constantes como

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

La mayoría de ellas son específicas para ciertos problemas matemáticos, especialmente si interviene la trigonometría o si hay que usar logaritmos o exponentiales. 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_03_02_07a.cs
// Ejemplo de funciones trigonométricas
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_07a
{
    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:

(3.2.7.1) Crea un programa que halle (y muestre) la raíz cuadrada del número que introduzca el usuario. Se repetirá hasta que introduzca 0.

(3.2.7.2) Diseña un programa que calcule 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.

(3.2.7.3) Crea un programa que calcule la distancia entre dos puntos (x_1, y_1) y (x_2, y_2) , usando la expresión $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

(3.2.7.4) 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 180 grados = π radianes.

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 asigne 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;

class Ejemplo_03_03_01a
{
    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 letras alternas (una sí y una no) 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". Estos caracteres especiales se preceden con una barra invertida (\). 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, y usar WriteLine como hemos hecho hasta ahora, porque \\n puede no funcionar correctamente en todos los sistemas operativos; más adelante veremos una alternativa más segura).

Estas secuencias especiales son las siguientes:

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

Vamos a ver un ejemplo que use las más habituales:

```
// Ejemplo_03_03_02a.cs
// Secuencias de escape
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_03_02a
{
    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: \", simples ', y barra \\\"");
    }
}
```

Su resultado sería este:

Esta es una frase

y esta es otra, separada dos lineas

Juguemos mas:

otro salto
Comillas dobles: ", 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, como alternativa, se puede usar una **arroba** (@) antes del texto (e incluso de las comillas), en vez de usar las barras invertidas:

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

Con este formato, el problema está si aparecen comillas en medio de la cadena. Para solucionarlo, se duplcan 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 básico de "string"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_04a
{
    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)** Crea 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)** Crea 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, tantas veces como sea necesario hasta que coincidan.

3.5. Los valores "booleanos"

En C# disponemos también de 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 ayudará a 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;

class Ejemplo_03_05a
{
    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 una versión alternativa del ejercicio 3.5.1, que asigne directamente el valor al booleano a partir de una comparación.
- (3.5.4)** 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.
- (3.5.5)** Crea una versión alternativa del ejercicio 3.5.4, que use "if" en vez del operador condicional.
- (3.5.6)** Crea una versión alternativa del ejercicio 3.5.5, que asigne directamente el valor al booleano a partir de una comparación.

3.6. Constantes y enumeraciones

En ocasiones, manejaremos valores que realmente no van a variar. Esto podría ocurrir, por ejemplo, con el número Pi, frecuente en matemáticas. 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 así el compilador no permitirá que la modifiquemos más adelante por error. Por convenio, para que sea fácil distinguir una constante de una variable, se suele escribir su nombre totalmente en mayúsculas:

```
const double PI = 3.1415926535;
```

Así, podríamos calcular la longitud de un circunferencia de esta forma :

```
// Ejemplo_03_06a.cs
// Constantes: longitud de una circunferencia
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_06a
{
    static void Main()
    {
        const double PI = 3.1415926535;
        double radio;

        Console.Write("Introduce el radio de la circunferencia: ");
        radio = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("La longitud es {0}", 2 * PI * radio);
    }
}
```

(Como curiosidad, Pi ya está definido en C#. Más adelante veremos las funciones y constantes existentes que están relacionadas con las matemáticas).

Ejercicios propuestos

(3.6.1) Crea un programa que permita convertir de millas a metros. El valor necesario para la conversión debe estar almacenado en una constante.

Cuando tenemos varias constantes, cuyos valores son números enteros, podemos dar los valores uno por uno, así:

```
const int LUNES = 0, MARTES = 1,
MIERCOLES = 2, JUEVES = 3,
VIERNES = 4, SABADO = 5,
DOMINGO = 6;
```

Pero también existe 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_03_06b.cs
// Ejemplo de enumeraciones
// Introducción a C#, por Nacho Cabanes
```

```

using System;

class Enumeraciones
{
    enum diasSemana { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES,
                      SABADO, DOMINGO };

    static void Main()
    {
        Console.Write("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, 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, tiene el valor: 2

Nota 1: 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 utilicen sus valores (como en `diasSemana.MIERCOLES`), mientras que en C se usa sólo el valor (MIERCOLES).

Nota 2: como puedes observar, las enumeraciones se deben declarar fuera de "Main".

Ejercicios propuestos

(3.6.2) Crea una enumeración para los meses del año, desde ENERO (con valor 1) hasta DICIEMBRE (con valor 12). Muestra el valor numérico correspondiente a OCTUBRE.

3.7. 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_03_07a.cs
// Ejemplo de uso de "var"
// Introducción a C#, por Nacho Cabanes

using System;

class UsoVar
{
    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

(3.7.1) 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".

Nota importante: Como puedes imaginar, el uso de "var", que es interesante que conozcas, no estará permitido en la mayoría de ejercicios de clase, en los que deberás saber qué tipo exacto de variable debes emplear, como parte de tu aprendizaje. Sólo se te permitirá utilizar "var" en el caso de que se te indique de forma explícita.

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 e indicando la posición del dato individual que nos interesa.

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

```
int[] ejemplo;
```

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

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

Si sabemos el tamaño desde el principio (algo que no siempre ocurrirá), podemos reservar espacio a la vez que declaramos la variable:

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

Es posible acceder a cada uno de los valores individuales indicando el nombre del array (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;

class Ejemplo_04_01_01a
{
    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 desde un primer momento. En ese 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;

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

```

```

    // Un entero que guardará su suma
    int 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 el que se muestra a continuación, 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 precedente, casi siempre se podrá abbreviar, 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];

```

Nota: En estos primeros ejemplos, usaremos "i<=4" para enfatizar que, en un array de 5 datos, el último de ellos tiene la posición 4, porque se empieza a contar desde

cero. Aun así, es más habitual comparar con el tamaño, haciendo "i<5", de modo que las órdenes anteriores se podrían escribir también (y es la manera que más usaremos dentro de poco):

```
suma = 0;           // Valor inicial de la suma: 0
for (int i = 0; i < 5; i++) // Y calculamos la suma repetitiva
    suma += numeros[i];
```

El fuente completo podría ser:

```
// 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;

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

        suma = 0;           // Valor inicial de la suma: 0
        for (int i = 0; i < 5; 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, si los datos iniciales no están prefijados, lo habitual será **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;

class Ejemplo_04_01_03b
{
    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". En el siguiente apartado verás detalles de cómo hacer ese tipo de búsquedas.

(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 (la raíz cuadrada de la suma de sus componentes al cuadrado; por ejemplo, para [2, -5, 7] el resultado sería la raíz cuadrada de 78, aproximadamente 8,83).

(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 ($x_1 \cdot y_1 + x_2 \cdot y_2 + z_1 \cdot z_2$).

(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. Si no lo consigues, en el próximo apartado tienes más detalles sobre cómo resolver este problema.

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

Algunas operaciones con datos pertenecientes a un array son especialmente frecuentes: buscar si existe un cierto dato, localizar el máximo o el mínimo, 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 cada 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.

```
for (i=0; i < cantidad; i++)
    if (datos[i] == 15)
        encontrado = true;
```

En ese caso, partiríamos del supuesto inicial de que el dato no existe (`encontrado = false`). Un planteamiento alternativo incorrecto es usar "else":

```
for (i=0; i < cantidad; i++) // Búsqueda incorrecta
    if (datos[i] == 15)
        encontrado = true;
    else
        encontrado = false; // Error!
```

Al salir de este bucle "for" no se nos diría realmente si el dato existe o no, sino si el dato está en la última posición. Por eso, se debe presuponer que el dato no existe y la búsqueda sólo debe comprobar si el dato sí aparece.

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 actualizarlo si fuera necesario.

```
int maximo = datos[0];
for (i=1; i < cantidad; i++)
    if (datos[i] > maximo)
        maximo = datos[i];
```

Nuevamente, un planteamiento incorrecto habitual es dar el valor inicial 0 al máximo (o mínimo). Esta aproximación fallaría si todos los datos son negativos:

```
int maximo = 0; // Error!
for (i=1; i < cantidad; i++)
    if (datos[i] > maximo)
        maximo = datos[i];
```

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. Es lo que se conoce como un "**array sobredimensionado**".

```
if (cantidad < capacidad)
{
    datos[cantidad] = 6;
    cantidad++;
}
```

En un caso real, y con un lenguaje moderno como C#, en general no será necesario emplear arrays sobredimensionados, porque tendremos a nuestra disposición otras estructuras dinámicas (capaces de "crecer" cuando sea necesario) como las listas, que estudiaremos más adelante.

Para **insertar** un dato en una cierta posición de un array sobredimensionado, los que vayan a quedar situados 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 (hay un dato más).

```
for (i=cantidad; i > posicionInsertar; i--)
    datos[i] = datos[i-1];
datos[posicionInsertar] = 30;
cantidad++;
```

Si se quiere **borrar** el dato que hay en una cierta posición de un array sobredimensionado, 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 (tenemos un dato menos).

```
int posicionBorrar = 1;
for (i=posicionBorrar; i < cantidad-1; i++)
    datos[i] = datos[i+1];
cantidad--;
```

Vamos a ver todo ello en un ejemplo completo:

```
// Ejemplo_04_01_04a.cs
// Añadir y borrar en un array
// Introducción a C#, por Nacho Cabanes

using System;

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

        int capacidad = 5;           // Capacidad máxima 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 15 12 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 (como ya hemos anticipado, 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), como muestra el siguiente ejemplo:

```

// Ejemplo_04_01_04b.cs
// Búsqueda incorrecta y correcta en un array
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_01_04b
{
    static void Main()
    {
        int[] datos = {10, 15, 12, 23, -5};
        int cantidad = 5;
        int i;

        // Buscamos el dato "23", de forma incorrecta
        Console.WriteLine("Búsqueda incorrecta:");
        for (i=0; i<cantidad; i++)
            if (datos[i] != 23)
                Console.WriteLine("No existe 23");
            else
                Console.WriteLine("Existe 23");

        // Buscamos -6 y 12, de forma correcta
        Console.WriteLine();
        Console.WriteLine("Búsqueda correcta:");
        bool encontrado;

        encontrado = false;
        for (i=0; i<cantidad; i++)
            if (datos[i] == -6)
                encontrado = true;
        if (encontrado)
            Console.WriteLine("Existe -6");
        else
            Console.WriteLine("No existe -6");

        encontrado = false;
        for (i=0; i<cantidad; i++)
            if (datos[i] == 12)
                encontrado = true;
        if (encontrado)
            Console.WriteLine("Existe 12");
        else
    }
}

```

```

        Console.WriteLine("No existe 12");

    }
}

```

Otra alternativa es emplear un contador que nos permita saber cuántas veces aparece ese dato en el array:

```

// Ejemplo_04_01_04c.cs
// Búsqueda usando un contador
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_01_04c
{
    static void Main()
    {
        int[] datos = {10, 15, 12, 23, -5};
        int cantidad = 5;
        int i;

        int veces = 0;

        for (i=0; i<cantidad; i++)
            if (datos[i] == 12)
                veces++;

        if (veces == 0)
            Console.WriteLine("No existe el dato");
        else
            Console.WriteLine("Sí existe el dato");
    }
}

```

Ejercicios propuestos:

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

(4.1.4.5) Crea un programa que准备 espacio para un máximo de 10 nombres. Deberá mostrar al usuario un menú que le permita realizar las siguientes operaciones:

- Añadir un dato al final de los ya existentes.
- Insertar un dato en una cierta posición (como ya se ha comentado, los que queden detrás deberán desplazarse "hacia el final" para dejarle hueco); por ejemplo, si el array contiene "hola", "adiós" y se pide insertar "bien" en la segunda posición, el array pasará a contener "hola", "bien", "adiós".
- Borrar el dato que hay en una cierta posición (como se ha visto, lo que estaban detrás deberán desplazarse "hacia el principio" para que no haya huecos); por ejemplo, si el array contiene "hola", "bien", "adiós" y se pide borrar el dato de la segunda posición, el array pasará a contener "hola", "adiós"
- Mostrar los datos que contiene el array.
- Salir del programa.

4.1.5. Constantes y tamaño del array

Hemos visto cómo declarar que un dato va a ser "constante", mediante el modificador "const", para evitar que su valor pueda ser alterado accidentalmente.

En el caso de un array sobredimensionado, el tamaño máximo también será constante, de modo que puede resultar más legible y hacer el programa más fácil de mantener si no empleamos una cifra numérica sino una constante con nombre.

Así, una nueva versión del fuente del apartado 4.1.3 (b), usando una constante llamada MAXIMO 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;

class Ejemplo_04_01_05a
{
    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);
    }
}

```

Si el array no está sobredimensionado y lo vamos a recorrer en todo su tamaño, no sería necesario utilizar una constante, sino que podemos saber su longitud añadiendo **".Length"** a su nombre, como en este ejemplo:

```

// Ejemplo_04_01_05b.cs
// Tamaño de un array: .Length
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_01_05b
{
    static void Main()
    {

        int[] numeros = new int[5];
        int suma;

        for (int i=0; i < numeros.Length; 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 < numeros.Length; 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.

(4.1.5.2) Crea una nueva versión del ejercicio 4.1.5.1, usando **".Length"** en vez de una constante.

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 o 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 dentro de poco en un segundo ejemplo.

En cualquier caso, si queremos indicar valores iniciales, lo haremos entre llaves, de forma parecida a como haríamos si fuera una tabla de una única dimensión, como veremos en el próximo ejemplo.

Vamos a ver una primera muestra de uso con **arrays "rectangulares"**, de la forma [2,20], lo que podríamos llamar el "estilo Pascal" (porque es la sintaxis que se emplea en ese lenguaje de programación). 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;

class Ejemplo_04_02a
{
    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 representar 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.

Si queremos recorrer por completo un array de varias dimensiones, no nos bastará con ".Length", sino que querremos saber la cantidad de filas y de columnas, o, en general, de datos que hay en cada una de las dimensiones. Para eso, podemos emplear "**.GetLength(n)**", donde n será un número desde 0 (la primera dimensión, típicamente las filas) hasta n-1 (la última dimensión, que serán las columnas en el caso de dos dimensiones):

```

// Ejemplo_04_02a2.cs
// Array de dos dimensiones "rectangulares" + GetLength
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_02a2
{
    static void Main()
    {
        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}
        };

        for (int i = 0; i < notas2.GetLength(0); i++)
        {
            for (int j = 0; j < notas2.GetLength(1); j++)
            {
                Console.Write("{0} ", notas2[i,j]);
            }
            Console.WriteLine();
        }
    }
}

```

```

    }
}

```

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, como ya sabemos, se puede emplear ".**Length**":

```

// Ejemplo_04_02b.cs
// Array de arrays (array de dos dimensiones al estilo C)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_02b
{
    static void Main()
    {
        int[][] notas;           // Array de dos dimensiones
        notas = new int[3][];    // Serán 3 bloques de datos
        notas[0] = new int[10];   // 10 notas en un grupo
        notas[1] = new int[15];   // 15 notas en otro grupo
        notas[2] = new int[12];   // 12 notas en el 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 matricial**:

- (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 de 4x4 son linealmente dependientes.
- (4.2.7)** Un programa que use matrices para resolver un sistema de ecuaciones lineales (por ejemplo, de 3 ecuaciones con 3 incógnitas) 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 primer lugar, deberemos declarar cual va a ser la estructura interna de nuestro registro, lo que no se puede hacer dentro de "Main". Más adelante, ya dentro del cuerpo del programa, podremos declarar variables de ese nuevo tipo.

Los datos que forman un "struct" pueden ser públicos o privados, como veremos posteriormente con más detalle. Por ahora, a nosotros nos interesará que sean accesibles desde el resto de nuestro programa, por lo que siempre los precederemos con la palabra "**public**" para indicar que queremos que sean públicos. Más adelante hablaremos de los demás "especificadores de acceso" que existen y de lo que supone cada uno de ellos.

Ya dentro del 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 campo, separados por un punto:

```
// Ejemplo_04_03_01a.cs

// Registros (struct)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_03_01a
{
    struct tipoPersona
    {
        public string nombre;
        public char inicial;
        public int edad;
        public float nota;
    }

    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 hasta 100 personas podríamos hacer:

```
// Ejemplo_04_03_02a.cs
// Array de struct
// Introducción a C#, por Nacho Cabanes
```

```

using System;

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

    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);
    }
}

```

En este ejemplo, para un array de 100 datos, 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 al usuario realizar 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). Recuerda que el array estará sobredimensionado, así que deberás llevar un contador de la cantidad de datos que hay almacenados hasta el momento.

(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 (mostrando entonces todos sus datos).

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. Es lo que se conoce como un "**struct anidado**". Por ejemplo, una fecha de nacimiento podría estar formada por día, mes y año. Para hacerlo desde C#, incluiríamos un "struct" en la definición del de otro, así:

```
// Ejemplo_04_03_03a.cs
// Registros anidados
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_03_03a
{
    struct fechaNacimiento
    {
        public byte dia;
        public byte mes;
        public short anyo;
    }

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

    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.WriteLine), así como la forma de darles un valor (=) y de comparar cual es su valor (==).

Vamos a comenzar por 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;

class Ejemplo_04_04_01a
{
    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 == "Héctor")
            Console.WriteLine("Dices que eres Héctor?");
        else
            Console.WriteLine("Así que no eres Héctor?");

        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 y el 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 acceder a 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 será necesario utilizar 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;

class Ejemplo_04_04_02a
{
    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

Se puede saber cuántas letras forman una cadena con "cadena.Length". Si contiene n letras, la primera estará en la posición 0 y la última estará en la posición n-1. 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;

class Ejemplo_04_04_03a
{
    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 50 cifras), que se deberán pedir como cadena de texto y analizar letra a letra (pista: tendrás que pensar cómo sumas dos números a mano: qué ocurre si al suma cifra a cifra obtienes un número mayor que 10 y cómo tratar el caso de que los dos números no tengan la misma longitud).

(4.4.3.5) Un programa capaz de multiplicar dos números enteros muy grandes (por ejemplo, de 50 cifras), que se deberán pedir como cadena de texto y analizar letra a letra (pista: nuevamente, deberás pensar cómo lo haces en papel:

multiplicar cifra a cifra, desplazar cuando terminas una cifra... te ayudará también haber hecho el ejercicio anterior para poder sumar dos números grandes).

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;

class Ejemplo_04_04_04a
{
    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
```

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

```
n
```

```

an
uan
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.WriteLine("Bienvenido, Juan");
```

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

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

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

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

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

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** y **Contains** podría ser:

```

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

using System;

class Ejemplo_04_04_05a
{
    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));
    }
}
```

```

    if (saludo.Contains(subcadena))
        Console.WriteLine("Efectivamente, aparece");
    else
        Console.WriteLine("No aparece");
}
}

```

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 informe además de 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;

class Ejemplo_04_04_06a
{
    static void Main()
    {
        string nombre = "Juan Pérez";
        string apellido = "Pérez";
        string nombreCompleto = nombre + " " + apellido;
        string nombreCorrecto = nombre.ToUpper();
        string password = "123456";
        string password2 = password.ToLower();
        string subcadena = "a";
        string textoASustituir = "Pepe";
        string cadenaSustituta = "Jose";
        string nombreFormal = nombre.Insert(0, "Don ");
        string apellidos = nombreCompleto.Remove(0, 6);
        string nombreCorregido = nombre.Replace("Pepe", "Jose");
    }
}

```

```

{
    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 posición de \"que\" es: {0}",
                      ejemplo.IndexOf("que"));

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

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

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

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

    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 esta 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 4.4.5.1 (buscar textos en un array de frases), 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;

class Ejemplo_04_04_07a
{
    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 emplear un conjunto de delimitadores (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;

class Ejemplo_04_04_07b
{
    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]);
    }
}

```

Si no se indica un delimitador en Split, se dará por sentado que sea deseado partir empleando espacios , así:

```

// Ejemplo_04_04_07c.cs
// Cadenas de texto: partir con "Split" (sin parámetros)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_04_07c
{
    static void Main()
    {
        string ejemplo = "uno dos tres cuatro";
        string [] ejemploPartido = ejemplo.Split();

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

```

Como curiosidad, también se puede dato el paso contrario: crear una cadena a partir de un separador y de un array de cadenas, con "String.Join" (cuidado: por motivos que veremos más adelante, "String" debe comenzar por mayúsculas):

```
// Ejemplo_04_04_07d.cs
// Cadenas de texto: partir con "Split" y unir con "Join"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_04_07d
{
    static void Main()
    {
        string ejemplo = "uno dos tres cuatro";
        string [] ejemploPartido = ejemplo.Split();
        Console.WriteLine( String.Join("-", ejemploPartido) );
    }
}
```

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 cómo ver si una cadena tiene exactamente un cierto valor, con el operador de igualdad (==), pero no sabemos comprobar qué cadena es "mayor" que otra (cuál aparecería la última de las dos en un diccionario), y se trata de algo que es necesario si deseamos ordenar textos. El operador "mayor que" (>), que usamos con los números, no se puede aplicar directamente a las cadenas. En su lugar, debemos emplear "CompareTo", que devolverá un número mayor que 0 si 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 (o minúsculas) 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)");

// Forma alternativa, con .ToUpper()
if (frase.ToUpper().CompareTo("hola".ToUpper()) > 0)
    Console.WriteLine("Es mayor que hola (mays o mins, forma 2)");
```

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;

class Ejemplo_04_04_08a
{
    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)");

        // Forma alternativa, con .ToUpper()
        if (frase.ToUpper().CompareTo("hola".ToUpper()) > 0)
            Console.WriteLine("Es mayor que hola (mays o mins 2)");
        else if (frase.ToUpper().CompareTo("hola".ToUpper()) < 0)
            Console.WriteLine("Es menor que hola (mays o mins 2)");
        else
            Console.WriteLine("Es hola (mays o mins 2)");
    }
}
```

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)
Es menor que hola (mays o mins 2)
```

Si escribimos "hOLa", que coincide con "hola" salvo por las mayúsculas, una comparación normal nos dirá que es mayor (en C#, las mayúsculas se consideran "mayores" que las minúsculas, aunque el criterio puede ser distinto en otros lenguajes de programación), 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)
Es hola (mays o mins 2)
```

Ejercicios propuestos:

(4.4.8.1) Un programa que pida al usuario dos frases y diga cuál 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 operaciones 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 los "StringBuilder", que sí lo permiten pero son algo más complejos de manejar: hay que 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

class Ejemplo_04_04_09a
{
    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) Prepara 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.

4.4.10. Cadenas repetitivas

En ocasiones, tendremos la necesidad de crear una cadena repetitiva, formada por un mismo carácter varias veces. Con nuestros conocimientos actuales, podríamos hacerlo concatenando a partir de una cadena vacía, así:

```
string asteriscos = "";
for (int i=0; i<10; i++)
    asteriscos += '*';
```

Pero también existe la posibilidad de usar "new String", indicando el carácter que queremos repetir y la cantidad de veces, de esta forma:

```
string asteriscos = new String('*', 10);
```

Ejercicios propuestos:

(4.4.10.1) Crea un programa que pida al usuario una frase y la muestre subrayada, usando para ello una cadena formada por tantos guiones como letras tuviera la frase inicial.

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), y con la que ya tuvimos un contacto en el apartado 2.4. Se trata de "foreach".

Se usa con el formato "foreach (variable in ConjuntoDeValores)". Será útil cuando queramos obtener todos los elementos del array o cadena, pero sin preocuparnos la posición en la que se encuentran:

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

using System;

class Ejemplo_04_05a
{
    static void Main()
    {
        int[] diasMes = {31, 28, 31};
        foreach(int dias in diasMes) {
            Console.WriteLine("Días del mes: {0}", dias);
        }

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

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

La orden "foreach" también se puede aplicar a arrays bidimensionales, si estamos en el caso de que nos interese obtener todos los datos que contiene y no necesitemos saber de qué posición procede cada dato, como en este ejemplo:

```
// Ejemplo_04_05b.cs
// Ejemplo de "foreach" con arrays bidimensionales
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_05b
{
    static void Main()
    {
        int[,] datos =
        {
            {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
            {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
        };

        foreach (int n in datos)
        {
            Console.Write("{0} ", n);
        }
    }
}
```

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), empleando "foreach".

4.6. Ejemplo completo

Vamos a plantear 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 aún no sabemos usar 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.

El array no estará completamente lleno, sino sobredimensionado: habrá espacio para 1000 datos, pero iremos añadiendo de uno en uno. Deberemos 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#, y, por eso, será la que empleemos en este programa).

El resto 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;

class Ejemplo_04_06a {

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

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

        do {
            // Menú 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 (int 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 qué tamaño quieres ver? ");
                    tamanyoBuscar = Convert.ToInt64( Console.ReadLine() );
                    for (int 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 (int 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 opción: no válida
        Console.WriteLine("Opción desconocida!");
        break;
    }
} while (opcion != 5); // Si la opción es 5, terminamos
}

```

(Como quizás 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**:

- En un caso real, es habitual que cada ficha tenga que guardar más información, no sólo esos dos campos de ejemplo que hemos previsto esta vez.
- Cuando nos muestra todos los datos en pantalla, si 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 detuviese cuando se llenase la pantalla de información (por ejemplo, una pausa tras mostrar cada 24 datos, ya que una pantalla de texto -consola- convencional tiene 25 líneas de texto).
- Por descontado, se nos pueden ocurrir muchas más preguntas que hacerle sobre nuestros datos: ¿qué fichas contienen un cierto texto? ¿ver los datos ordenados? ¿buscar entre dos tamaños? ¿buscar por datos adicionales, como la categoría a la que pertenecen? ...
- Además, cuando salimos del programa se borran todos los datos que habíamos tecleado, pero eso es lo único "casi inevitable", porque aún no sabemos manejar ficheros ni bases de datos.

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) Mejora 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) Amplía 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, por ejemplo).

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

(4.6.7) Amplía 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) Mejora 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) Amplía 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 necesitar ordenar datos que tenemos almacenados 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 (1.000×1.000) y en general para n datos se requerirán n^2 comparaciones.

Se pueden realizar ligeras mejoras (por ejemplo, cambiar uno de los "for" por un "while", para no repasar todos los datos si ya estaban parcialmente ordenados), y existen 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

(Consiste en 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 se busca el menor, y se 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

(Se trata de comparar cada elemento con los anteriores -que ya están ordenados- y desplazarlo hasta su posición correcta, usando "while" en vez de "for").

```

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

```

(Se puede mejorar, 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 de estos algoritmos podría ser:

```

// Ejemplo_04_07a.cs
// Ordenaciones simples
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_04_07a
{
    static void Main()
    {

        int[] datos = {5, 3, 14, 20, 8, 9, 1};
        int i,j,datoTemporal;
        int n=7; // Número de datos

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

```

```

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

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

// SELECCIÓN DIRECTA:
// (En cada pasada busca el menor,y lo interc. 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.WriteLine("{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.WriteLine("{0}", dato);
Console.WriteLine();

// INSERCIÓN DIRECTA:
// (Comparar cada elemento con los anteriores -ya 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.Write("{0} ",dato);
    Console.WriteLine();

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

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

}

```

Y su resultado sería:

Ordenando mediante burbuja...

```

5 3 14 20 8 9 1
1 5 14 20 8 9 3
1 3 14 20 8 9 5
1 3 5 20 14 9 8
1 3 5 8 20 14 9
1 3 5 8 9 20 14

```

Ordenado:1 3 5 8 9 14 20

Ordenando mediante selección directa...

```

5 3 14 20 8 9 1
1 3 14 20 8 9 5
1 3 14 20 8 9 5
1 3 5 20 8 9 14
1 3 5 8 20 9 14
1 3 5 8 9 20 14

```

Ordenado:1 3 5 8 9 14 20

Ordenando mediante inserción directa...

```

5 3 14 20 8 9 1
3 5 14 20 8 9 1
3 5 14 20 8 9 1
3 5 14 20 8 9 1
3 5 8 14 20 9 1

```

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

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 contenido 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" en vez de un numero.
- (4.7.4)** Amplía 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 si fuera menor), 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 binaria realizará 10 comparaciones o menos. Se podría implementar así:

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

using System;

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

```

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

// Luego los ordenamos mediante burbuja
Console.Write("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;
    }
    // O 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 y que conozcas ciertos algoritmos habituales...

```

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

using System;

class Ejemplo_04_07c
{
    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();
    }
}

```

Si el tipo de datos repetitivo es más complejo (como un struct, por ejemplo), también se podrá usar **Array.Sort**, pero será necesario indicarle cual será el criterio de ordenación, empleando "interfaces", que es algo que veremos más adelante.

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".

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 subproblemas, 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 más breve y más 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, que será descomponer el proyecto en varias "clases" que cooperan entre ellas).

Estos "trozos" de programa se suelen llamar "funciones", "procedimientos" o "subrutinas", según el lenguaje del que se trate. En el caso de C# y sus derivados (entre los que está C#), el nombre que más se emplea 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 "static **void**", cuyo significado iremos viendo muy pronto. Después, entre llaves indicaremos todos los pasos que queremos que realice ese "trozo de programa".

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

```
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:

```
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", de modo que el fuente completo sería así:

```
// Ejemplo_05_02a.cs
// Función "Saludar"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_02a
{

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

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

El flujo de un programa que contenga funciones ya no será tan claramente "de arriba a abajo", porque cada "llamada" a una función interrumpirá ese orden, y en

ese punto del programa se "saltará" a esa función y se seguirán los pasos que dicha función indique. Por ejemplo, el fuente anterior se comportaría igual que éste:

```
// Ejemplo_05_02b.cs
// Eliminando la función "Saludar"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_02b
{
    static void Main()
    {
        Console.WriteLine("Empezamos...");
        Console.Write("Bienvenido al programa ");
        Console.WriteLine("de ejemplo");
        Console.WriteLine("Espero que estés bien");
        Console.WriteLine("Nada más por hoy...");
    }
}
```

Como ejemplo (incompleto) más real, 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 comparten 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:

```
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 obtendríamos 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
// Función "EscribirNumeroReal"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_03a
{

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

    static void Main()
    {
        float x;

        x= 5.1f;
        Console.WriteLine("El primer número 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
// Función "EscribirSuma"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_03b
{

    static void EscribirSuma( int a, int b )
    {
        Console.Write( a+b );
    }

    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**.
- También por convenio, en C# los nombres de las funciones se suelen escribir comenzando con una letra **mayúscula** (especialmente si son públicas, detalle que veremos dentro de poco). 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 de asteriscos 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, en ese orden. 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.
- (5.3.4) Crea una función "EscribirRepetido", que reciba un carácter y un número, y escriba ese carácter tantas veces como indique ese número (todas ellas en la misma línea).
- (5.3.5) Crea una nueva versión de la función "DibujarRectangulo", que se apoye en la "EscribirRepetido" que acabas de crear.

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

Hasta ahora hemos creado funciones que escribían textos en pantalla y que no "devolvían" ningún resultado. Por eso, antes del nombre de la función escribíamos la palabra "void" (nulo), como ocurría hasta ahora con "Main" y como hemos hecho con nuestra función "Saludar".

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

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:

```
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 "static" hasta

que profundicemos un poco más y veamos el significado de esa palabra, en el próximo tema.

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
// Función "Cuadrado"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_04a
{
    static int Cuadrado ( int n )
    {
        return n*n;
    }

    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 permita saber cuál es el mayor de dos números reales así:

```
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", si bien puede resultar desaconsejable en funciones más grandes que esta, porque hará que sea más difícil seguir el flujo de ese fragmento de programa. Por eso, la función anterior se podría escribir también así, con un solo "return", alternativa que generalmente resultará más legible:

```

static float Mayor ( float n1, float n2 )
{
    float mayor = n1;
    if (n2 > n1)
        mayor = n2;
    return mayor;
}

```

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:

```

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, sin devolver ningún valor.

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 en ese caso 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, se tratará una "**variable global**", a la que se podrá acceder desde cualquier parte. Por ahora, para nosotros, una variable global deberá ser precedida por 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 conseguimos que cada parte del programa trabaje con sus propios datos, y ayudamos a evitar que un error en un fragmento 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 (05_04). 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**", como también veremos en el próximo tema.

Vamos a practicar el uso de variables locales con un ejemplo. Crearemos una función que calcule la potencia de un número entero (un número elevado a otro), así como el cuerpo del programa que la use. La forma de conseguir elevar un número a otro será realizando varias 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 podemos necesitar calcular operaciones 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;

class Ejemplo_05_05a
{
    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
    }

    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. Esta función deberá pedir al usuario que introduzca el valor, tantas veces como sea necesario, deberá volvérsele a pedir en caso de error, y deberá 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 "SumarCifras" que reciba un numero cualquiera y que devuelva como resultado la suma de sus dígitos. Por ejemplo, si el número fuera 123, la suma sería 6.

(5.5.6) Crea una función "DibujarTriángulo" 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

**

*

Adaptar un programa realizado anteriormente (y que tenga una cierta complejidad) para **dividirlo en funciones** supondrá decidir qué variables se deben compartir entre distintas funciones (y serán variables globales, precedidas por "static") y qué variables no es necesario compartir (que serán variables locales). Por ejemplo, el "ejemplo_04_06a" se podría convertir en algo como esto:

```
// Ejemplo_05_05b.cs
// Versión del Ejemplo_04_06a.cs ampliada usando funciones
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_05b
{
    struct tipoFicha
    {
        public string nombreFich; // Nombre del fichero
```

```

    public long tamanyo;           // El tamaño en KB
}

// Variables "globales"
static tipoFicha[] fichas;
static int numeroFichas;

static void MostrarMenu()
{
    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");
}

static int PedirOpcion()
{
    return Convert.ToInt32(Console.ReadLine());
}

static void Anyadir()
{
    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)! ");
}

static void MostrarTodos()
{
    for (int i = 0; i < numeroFichas; i++)
        Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
            fichas[i].nombreFich, fichas[i].tamanyo);
}

static void MostrarPorTamanyo()
{
    Console.WriteLine("¿A partir de qué tamaño quieres ver? ");
    long tamanyoBuscar = Convert.ToInt64(Console.ReadLine());
    for (int i = 0; i < numeroFichas; i++)
        if (fichas[i].tamanyo >= tamanyoBuscar)
            Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
                fichas[i].nombreFich, fichas[i].tamanyo);
}

static void MostrarUnDatos()
{
    Console.WriteLine("¿De qué fichero quieres ver todos los datos? ");
    string textoBuscar = Console.ReadLine();
    for (int i = 0; i < numeroFichas; i++)
        if (fichas[i].nombreFich == textoBuscar)
            Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
                fichas[i].nombreFich, fichas[i].tamanyo);
}

```

```

static void AvisarFin()
{
    Console.WriteLine("Fin del programa");
}

static void AvisarError()
{
    Console.WriteLine("Opción desconocida!");
}

static void Main()
{
    // Variables locales
    int opcion;

    // Valores iniciales a las variables compartidas
    fichas = new tipoFicha[1000];
    numeroFichas = 0;

    do
    {
        MostrarMenu();
        opcion = PedirOpcion();
        switch (opcion)
        {
            case 1: Anyadir(); break;
            case 2: MostrarTodos(); break;
            case 3: MostrarPorTamanyo(); break;
            case 4: MostrarUnDatos(); break;
            case 5: AvisarFin(); break;
            default: AvisarError(); break;
        }
    } while (opcion != 5);
}

```

En un ejemplo como éste, si aparece la palabra "**public**" delante de nuestras funciones, también deberíamos añadir esa palabra delante de nuestro "struct", para evitar un error de compilación que diga que se está utilizando un dato que no es público desde funciones que sí lo son. Si, como en nuestro caso, no hemos utilizado "public", ese problema no debería existir.

Como se ve en este ejemplo, es posible que en los tipos de datos definidos por nosotros (como ese "**struct**") debamos añadir también el especificador "public", para evitar un error de compilación que diga que se está utilizando un dato que no es público desde funciones que sí lo son.

Las variables globales se deben evitar tanto como sea posible. Por ejemplo, una variable "i" que controle bucles siempre debería ser local, porque no es un dato que se deba compartir entre funciones. Lo mismo ocurre en el ejemplo anterior con variables como "textoBuscar" o "tamanyoBuscar".

En lenguajes más antiguos, como C, se tendía a intentar que **nada fuera global**, para evitar errores difíciles de detectar, especialmente en programas formados

por varios fuentes. Ese tipo de problemas son menos frecuentes en lenguajes más modernos y más modulares, como C#, pero aun así se podría crear una versión alternativa del fuente anterior en la que ninguna variable fuera global, sino que desde Main se llamaría a las demás funciones pasándoles como parámetros los datos con los que deben trabajar, y, según el caso, recibiendo esos valores devueltos si han sido modificados, por ejemplo así:

```
// Ejemplo_05_05c.cs
// Versión del Ejemplo_05_05b.cs sin variables globales
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_05c
{
    struct tipoFicha
    {
        public string nombreFich; // Nombre del fichero
        public long tamanyo; // El tamaño en KB
    }

    static void MostrarMenu()
    {
        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");
    }

    static int PedirOpcion()
    {
        return Convert.ToInt32(Console.ReadLine());
    }

    static int Anyadir(tipoFicha[] fichas, int numeroFichas)
    {
        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)! ");
        }
        return numeroFichas;
    }

    static void MostrarTodos(tipoFicha[] fichas, int numeroFichas)
    {
        for (int i = 0; i < numeroFichas; i++)
        {
            Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
                fichas[i].nombreFich, fichas[i].tamanyo);
        }
    }
}
```

```

static void MostrarPorTamanyo(tipoFicha[] fichas, int numeroFichas)
{
    Console.WriteLine("¿A partir de que tamaño quieres ver?");
    long tamanyoBuscar = Convert.ToInt64(Console.ReadLine());
    for (int i = 0; i < numeroFichas; i++)
        if (fichas[i].tamanyo >= tamanyoBuscar)
            Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
                fichas[i].nombreFich, fichas[i].tamanyo);
}

static void MostrarUnDatos(tipoFicha[] fichas, int numeroFichas)
{
    Console.WriteLine("¿De qué fichero quieres ver todos los datos?");
    string textoBuscar = Console.ReadLine();
    for (int i = 0; i < numeroFichas; i++)
        if (fichas[i].nombreFich == textoBuscar)
            Console.WriteLine("Nombre: {0}; Tamaño: {1} KB",
                fichas[i].nombreFich, fichas[i].tamanyo);
}

static void AvisarFin()
{
    Console.WriteLine("Fin del programa");
}

static void AvisarError()
{
    Console.WriteLine("Opción desconocida!");
}

static void Main()
{
    // Variables locales
    tipoFicha[] fichas = new tipoFicha[1000];
    int numeroFichas = 0;
    int opcion;

    do
    {
        MostrarMenu();
        opcion = PedirOpcion();
        switch (opcion)
        {
            case 1:
                numeroFichas = Anyadir(fichas, numeroFichas);
                break;
            case 2: MostrarTodos(fichas, numeroFichas); break;
            case 3: MostrarPorTamanyo(fichas, numeroFichas); break;
            case 4: MostrarUnDatos(fichas, numeroFichas); break;
            case 5: AvisarFin(); break;
            default: AvisarError(); break;
        }
    } while (opcion != 5);
}
}

```

Como se puede observar, este fuente se parece en general mucho al anterior, salvo por el detalle de que se pasan parámetros a las funciones y que "Anyadir" cambia ligeramente su comportamiento, porque puede alterar o no el valor de "numeroFichas", así que ese valor se devuelve. Dentro de poco veremos otras formas de modificar valores de datos que se pasen como parámetro.

Ejercicios propuestos:

- (5.5.7) Crea una nueva versión del ejercicio 4.6.3 (datos de 50 personas), en la que cada una de las funcionalidades que permite el programa esté desglosada en su propia función independiente de Main (PedirDatos, MostrarTodos, MostrarPorEdad, MostrarPorInicial):

Un concepto relacionado con el uso de variables globales y locales es el de "**transparencia referencial**": es deseable que una función devuelva siempre los mismos resultados cuando se llama con los mismos parámetros, sin depender de los valores de variables globales o de otros datos externos a la función.

5.6. Los conflictos de nombres en las variables

¿Qué ocurre si damos el mismo nombre a dos variables locales distintas en dos zonas distintas del programa? 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;

class Ejemplo_05_06a
{
    static void CambiaN()
    {
        int n = 7;
        n++;
    }

    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 y sí se propagarán los cambios:

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

using System;

class Ejemplo_05_06b
{
    static int n = 7;

    static void CambiaN()
    {
        n++;
    }

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

5.7. Modificando parámetros

5.7.1. Paso de parámetros por valor

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_07_01a.cs
// Modificar una variable recibida como parámetro - acercamiento
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_07_01a
{
    static void Duplicar(int x)
    {
```

```

        Console.WriteLine(" El valor recibido vale {0}", x);
        x = x * 2;
        Console.WriteLine(" y ahora vale {0}", x);
    }

    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.

5.7.2. Paso de parámetros por referencia

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_07_02a.cs
// Modificar una variable recibida como parámetro - correcto
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_07_02a
{

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

```

```

        Console.WriteLine(" y ahora vale {0}", x);
    }

    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:

```
static void Intercambiar(ref int x, ref int y)
```

Ejercicios propuestos:

(5.7.2.1) 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.

(5.7.2.2) 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.

(5.7.2.3) Crea una función "MaxMinArray", que reciba un array de reales de doble precisión y devuelva el mayor valor almacenado en ese array y el menor, utilizando parámetros por referencia. Pruébala con un "Main" adecuado.

5.7.3. Parámetros de salida

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_07_03a.cs
// Parámetros "out"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_07_03a
{
    static void ObtenerDosPrimerasLetras(string cadena,
                                         out char l1, out char l2)
    {
        l1 = cadena[0];
        l2 = cadena[1];
    }

    static void Main()
    {
        char letra1, letra2;
        ObtenerDosPrimerasLetras("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 asigne valores iniciales a "letra1" y "letra2".

(Como se ve en este ejemplo, es habitual que si una función devuelve más de un valor, lo haga mediante parámetros "ref" o "out" y la función pase a ser "**void**". No se suele considerar buena política devolver un parámetro mediante "return" y otros mediante "ref", salvo quizás en el caso de que mediante "return" se devuelva un código de error para comprobar si el funcionamiento ha sido correcto).

Como curiosidad adicional, en versiones muy recientes de C# se puede declarar la variable directamente en la llamada a la función, como en el siguiente ejemplo (que funciona en Visual Studio 2017 y posteriores, pero no en Visual Studio 2015 ni en el compilador de línea de comandos de .Net que incluye –actualmente– Widows 10):

```
// Ejemplo_05_07_03b.cs
// Parámetros "out" declarados en la llamada
// Introducción a C#, por Nacho Cabanes

using System;
```

```

class Ejemplo_05_07_03a
{
    static void ObtenerDosPrimerasLetras(string cadena,
                                         out char l1, out char l2)
    {
        l1 = cadena[0];
        l2 = cadena[1];
    }

    static void Main()
    {
        ObtenerDosPrimerasLetras("Nacho", out char letra1, out char letra2);
        Console.WriteLine("Las dos primeras letras son {0} y {1}",
                          letra1, letra2);
    }
}

```

Ejercicios propuestos:

(5.7.3.1) Crea una función "Iniciales", similar a la del ejercicio 5.7.2.2, que reciba una cadena como "Nacho Cabanes" y devuelva las letras N y C (primera letra, y letra situada tras el primer espacio), pero esta vez usando parámetros de salida. Crea un "Main" que te permita comprobar que funciona correctamente.

(5.7.3.2) Crea una función "MaxMinArray", parecida a la del ejercicio 5.7.2.3, que reciba un array de reales de doble precisión y devuelva el mayor valor almacenado en ese array y el menor, utilizando parámetros de salida. Pruébala con un "Main" adecuado.

5.7.4. Una aplicación de los parámetros de salida: pedir números sin try-catch (TryParse)

Existen algunas funciones ya incorporadas en la biblioteca estándar de C# que emplean parámetros de salida. Una de ellas, que puede resultar útil en ciertas circunstancias, es `Int32.TryParse`, que puede ser una alternativa cómoda a encerrar dentro de un bloque `try-catch` una conversión de cadena a número (por ejemplo, con `Convert.ToInt32`) para evitar que se interrumpa el programa en caso de introducir un dato incorrecto.

El siguiente ejemplo compara ambas formas de trabajar:

```

// Ejemplo_05_07_04a.cs
// Uso de "TryParse"
// Introducción a C#, por Nacho Cabanes

using System;

class ParseNums
{
    static void Main()
    {
        string numStr = "123";
        int num;
        if (int.TryParse(numStr, out num))
            Console.WriteLine("Número válido: " + num);
        else
            Console.WriteLine("Número inválido");
    }
}

```

```

{
    int a;

    // Planteamiento con try-catch
    Console.Write("Introduce un entero: ");
    try
    {
        a = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Su valor es {0}", a);
    }
    catch (Exception)
    {
        Console.WriteLine("No es un número válido");
    }

    // Alternativa con TryParse
    Console.Write("Introduce otro entero: ");
    if (Int32.TryParse(Console.ReadLine(), out a))
        Console.WriteLine("Su valor es {0}", a);
    else
        Console.WriteLine("No es un número válido");
}
}

```

El formato de TryParse no sólo es más compacto, sino que además resulta más fácil de llevar a condiciones como un do-while, por lo que puede ser una alternativa más cómoda que try-catch para comprobaciones sencillas.

Ejercicios propuestos

(5.7.4.1) Crea un programa que te pida tu edad tantas veces como sea necesario, hasta que introduzcas un valor numérico aceptable.

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 escribir 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;

class Ejemplo_05_08a
{

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

```

```

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

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

}

```

Ejercicios propuestos:

(5.8.1) Crea una nueva versión del ejercicio 5.7.1, en el que la función "Intercambiar" esté declarada después de "Main".

5.9. 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;

class Ejemplo_05_10a
{
    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
    }

    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 y, en muchos de esos casos, ese problema se podrá expresar de forma recursiva. La recursividad resulta muy útil también en la programación de cierto tipo de juegos: la búsqueda de soluciones para muchos juegos "por turnos" se puede plantear como una función recursiva que explore posibles soluciones a partir del estado actual del juego.

Los ejercicios propuestos te ayudarán a descubrir otros ejemplos de situaciones relativamente sencillas en las que se puede aplicar la recursividad.

Ejercicios propuestos:

- (5.9.1)** 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).
- (5.9.2)** 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".
- (5.9.3)** 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).
- (5.9.4)** 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).
- (5.9.5)** 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.
- (5.9.6)** 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 letra de la sexta posición).
- (5.9.7)** Crea, tanto de forma recursiva como de forma iterativa, una función diga si una cadena de caracteres es simétrica (un palíndromo). Por ejemplo, "OSO", "RADAR" y "DABALEARROZALAZORRAELABAD" son palíndromos.
- (5.9.8)** Crea 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, se puede: 1) Dividir m por n para obtener el resto r ($0 \leq r < n$) ; 2) Si $r = 0$, el MCD es n; 3) Si el resto no es cero, el máximo común divisor es $MCD(n,r)$.
- (5.9.9)** 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.

(5.9.10) 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.

(5.9.11) 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 "pivot" (por ejemplo, el punto medio); luego se recoloca el array de modo que todos los elementos menores que el pivot 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.

5.10. 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 en línea de comandos se pueden leer desde C#. Se consigue indicando un parámetro especial en Main, un array de strings:

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

Para conocer el valor de 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, o bien con "foreach":

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

Por otra parte, si queremos que nuestro programa **se interrumpa** en un cierto punto (algo poco recomendable en general, porque dificulta el seguimiento del flujo del programa), 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):

```
static int Main(string[] args)
{
    ...
    return 0;
}
```

Un ejemplo que pusiera todo esto a prueba podría ser:

```
// Ejemplo_05_10a.cs
// Parámetros y valor de retorno de "Main" (1)
// Introducción a C#, por Nacho Cabanes

using System;
```

```

class Ejemplo_05_10a
{
    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;
    }
}

```

O, de forma alternativa::

```

// Ejemplo_05_10b.cs
// Parámetros y valor de retorno de "Main" (2)
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_10b
{
    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!");
            return 1;
        }
        else
            return 0;
    }
}

```

Ejercicios propuestos:

(5.10.1) 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.

(5.10.2) 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".

(5.10.3) Crea una variante del ejercicio 5.10.2, en la que Main devuelva el código 1 si la operación indicada no es válida o 0 cuando sí sea una operación aceptable.

(5.10.4) Crea una variante del ejercicio 5.10.3, 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.

5.11. Parámetros con valores por defecto y con nombre

Cada vez más lenguajes permiten indicar parámetros con "valor por defecto", para los que es posible indicar un valor en la llamada o bien no detallarlo y utilizar el valor previsto en la declaración de la función. En el caso de C# es posible usarlos desde la versión 4, del año 2010. Ese tipo de parámetros, si los hay, deberán ser los últimos de la lista:

```
// Ejemplo_05_11a.cs
// Parámetros por defecto
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_11a
{
    static string Linea(int veces, char letra='*')
    {
        return new string(letra, veces);
    }

    static void Main()
    {
        Console.WriteLine( Linea(10, '-' ) );
        Console.WriteLine( Linea(12) );
    }
}
```

Otra posibilidad avanzada (y reciente, y que existe en pocos lenguajes) es la de indicar los parámetros en un orden distinto al que se definieron. Para eso, en C# se debe preceder cada parámetro por su nombre, usando un símbolo de "dos puntos" como separador:

```
// Ejemplo_05_11b.cs
// Parámetros con nombre
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_05_11b
{
    static string Linea(int veces, char letra='*')
    {
        return new string(letra, veces);
    }

    static void Main()
    {
        Console.WriteLine( Linea(letra: '=', veces: 8) );
    }
}
```

Ejercicios propuestos:

(5.11.1) Crea una función "DibujarRecuadro", que muestre en pantalla un recuadro del ancho y alto que indique el usuario, y relleno con el carácter que también se indique como tercer parámetro. El carácter será opcional, y se usarán "almohadillas" si no se indicar otro distinto. El alto también será opcional, con un valor por defecto de 3. Pruébalo desde Main.

(5.11.2) Crea un segundo programa que use la función "DibujarRecuadro", en esta ocasión llamando con los parámetros en orden inverso (primero el carácter, luego el alto y finalmente el ancho).