

1. Toma de contacto con C#

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

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

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

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

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

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

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

1.1 Escribir un texto en C#

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

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

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

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

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

Ejercicio propuesto (1.1.1): Crea un programa en C# que te salude por tu nombre (ej: "Hola, Nacho").

Sólo un par de cosas más antes de seguir adelante:

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

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

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

```
public class Ejemplo01 {
```

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

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

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

1.2. Cómo probar este programa

1.2.1 Cómo probarlo con Mono

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

Podemos descargar Mono desde su página oficial:

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



En la parte superior derecha aparece el enlace para descargar ("download now"), que nos lleva a una nueva página en la que debemos elegir la plataforma para la que queremos nuestro Mono. Nosotros descargaremos la versión más reciente para Windows (la 2.10.5 en el momento de escribir este texto).

```
public class Ejemplo01b
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
        System.Console.ReadLine();
    }
}
```

1.3. Mostrar números enteros en pantalla

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

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

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

Ejercicios propuestos:

- **(1.3.1)** Crea un programa que diga el resultado de sumar 118 y 56.
- **(1.3.2)** Crea un programa que diga el resultado de sumar 12345 y 67890.

1.4. Operaciones aritméticas básicas

1.4.1. Operadores

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

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

Ejercicios propuestos:

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

1.4.2. Orden de prioridad de los operadores

Sencillo:

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

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

- **(1.4.2.1)** Calcular el resultado de $-2 + 3 * 5$
- **(1.4.2.2)** Calcular el resultado de $(20+5) \% 6$
- **(1.4.2.3)** Calcular el resultado de $15 + -5*6 / 10$
- **(1.4.2.4)** Calcular el resultado de $2 + 10 / 5 * 2 - 7 \% 1$

1.4.3. Introducción a los problemas de desbordamiento

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

```
public class Ejemplo01multiplic
{
    public static void Main()
    {
        System.Console.WriteLine(10000000*10000000);
    }
}
```

1.5. Introducción a las variables: int

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

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

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

1.5.1. Definición de variables: números enteros

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

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

```
int primerNumero;
```

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

1.5.2. Asignación de valores

Podemos darle un valor a esa variable durante el programa haciendo

```
primerNumero = 234;
```

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

```
int primerNumero = 234;
```

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

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

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

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

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

1.5.3. Mostrar el valor de una variable en pantalla

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

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

pero si se trata de una variable es idéntico:

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

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

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

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

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

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

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

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

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

Repasemos lo que hace:

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

- `System.Console.WriteLine("La suma de {0} y {1} es {2}", primerNumero, segundoNumero, suma);` muestra en pantalla el texto y los valores de las tres variables (los dos números iniciales y su suma).

Ejercicios propuestos:

- **(1.5.3.1)** Crea un programa que calcule el producto de los números 121 y 132, usando variables.
- **(1.5.3.2)** Crea un programa que calcule la suma de 285 y 1396, usando variables.
- **(1.5.3.3)** Crea un programa que calcule el resto de dividir 3784 entre 16, usando variables.

1.6. Identificadores

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

Por eso, no son nombres de variable válidos:

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

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

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

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

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

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

```
int primerNumero;
```

1.7. Comentarios

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

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

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

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

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

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

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

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

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

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

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

1.8. Datos por el usuario: `ReadLine`

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

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

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

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

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

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

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

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

Ejercicios propuestos:

- **(1.8.1)** Crea un programa que calcule el producto de dos números introducidos por el usuario.
- **(1.8.2)** Crea un programa que calcule la división de dos números introducidos por el usuario, así como el resto de esa división.

1.9. Pequeñas mejoras

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

```
using System;

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

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

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

```
}
```

Podemos declarar varias variables a la vez, si van a almacenar datos del mismo tipo. Para hacerlo, tras el tipo de datos indicaríamos todos sus nombres, separados por comas:

```
using System;

public class Ejemplo04b
{
    public static void Main()
    {
        int primerNumero, segundoNumero, suma;

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

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

Y podemos escribir sin avanzar a la línea siguiente de pantalla, si usamos "Write" en vez de "WriteLine":

```
using System;

public class Ejemplo04c
{
    public static void Main()
    {
        int primerNumero, segundoNumero, suma;

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

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

Y ahora que conocemos los fundamentos, puede ser el momento de pasar a un editor de texto un poco más avanzado. Por ejemplo, en Windows podemos usar Notepad++, que es gratuito, destaca la sintaxis en colores, muestra la línea y columna en la que nos encontramos, ayuda a encontrar las llaves emparejadas, realza la línea en la que nos encontramos, tiene soporte para múltiples ventanas, etc.:

2. Estructuras de control

2.1. Estructuras alternativas

2.1.1. if

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

```
if (condición) sentencia;
```

Vamos a verlo con un ejemplo:

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

using System;

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

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

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

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

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

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

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

Ejercicios propuestos:

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

2.1.2. if y sentencias compuestas

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

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

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

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

En este caso, si el número es positivo, se hacen dos cosas: escribir un texto y luego... ¡escribir otro! (En este ejemplo, esos dos "WriteLine" podrían ser uno solo, en el que los dos textos estuvieran separados por un carácter especial, el símbolo de "salto de línea"; más adelante

iremos encontrando casos en lo que necesitemos hacer cosas "más serias" dentro de una sentencia compuesta).

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

Ejercicios propuestos:

- **(2.1.2.1)** Crear un programa que pida al usuario un número entero. Si es múltiplo de 10, se lo avisará al usuario y pedirá un segundo número, para decir a continuación si este segundo número también es múltiplo de 10.

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

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

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

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

```
/*
/* Ejemplo en C# nº 7: */
/* ejemplo07.cs */
/*
/*
/* Condiciones con if (3) */
/*
/*
/* Introduccion a C#,
/* Nacho Cabanes */
/*
using System;

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

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

```

    if (numero != 0)
        Console.WriteLine("El número no es cero.");
}

```

Ejercicios propuestos:

- **(2.1.3.1)** Crear 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)** Crear un programa que pida al usuario dos números enteros. Si el segundo no es cero, mostrará el resultado de dividir entre el primero y el segundo. Por el contrario, si el segundo número es cero, escribirá "Error: No se puede dividir entre cero".

2.1.4. if-else

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

```

/*
 *----- Ejemplo en C# nº 8: -----
 *----- ejemplo08.cs -----
 *----- -----
 *----- Condiciones con if (4) -----
 *----- -----
 *----- Introducción a C#,
 *----- Nacho Cabanes -----
 *----- -----
 */

using System;

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

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

```

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

```

/*
 *----- Ejemplo en C# nº 9: -----
 *----- ejemplo09.cs -----
 *----- -----
 */

```

```

/*
/* Condiciones con if (5) */
/*
/* Introduccion a C#, */
/* Nacho Cabanes */
/*-----*/
using System;

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

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

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

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

```

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

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

```

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

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

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

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

```

```

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

```

Ejercicio propuesto:

- **(2.1.4.1)** Mejorar la solución al ejercicio 2.1.3.1, usando "else".
- **(2.1.4.2)** Mejorar la solución al ejercicio 2.1.3.2, usando "else".

2.1.5. Operadores lógicos: &&, ||, !

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

Operador	Significado
&&	Y
	O
!	No

De modo que podremos escribir cosas como

```

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

```

Ejercicios propuestos:

- **(2.1.5.1)** Crear un programa que pida al usuario un número enteros y diga si es múltiplo de 2 o de 3.
- **(2.1.5.2)** Crear un programa que pida al usuario dos números enteros y diga "Uno de los números es positivo", "Los dos números son positivos" o bien "Ninguno de los números es positivo", según corresponda.
- **(2.1.5.3)** Crear un programa que pida al usuario tres números reales y muestre cuál es el mayor de los tres.
- **(2.1.5.4)** Crear un programa que pida al usuario dos números enteros cortos y diga si son iguales o, en caso contrario, cuál es el mayor de ellos.

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

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

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

En el caso del lenguaje C#, este riesgo no existe, porque la "condición" debe ser algo cuyo resultado "verdadero" o "falso" (un dato de tipo "bool"), de modo que obtendríamos un error de

compilación "Cannot implicitly convert type 'int' to 'bool'" (*no puedo convertir un "int" a "bool"*). Es el caso del siguiente programa:

```
/*
 *----- Ejemplo en C# nº 11: ejemplo11.cs -----
 *----- Condiciones con if (7) -----
 *----- comparacion incorrecta -----
 *----- Introduccion a C#, Nacho Cabanes -----
 */----- */

using System;

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

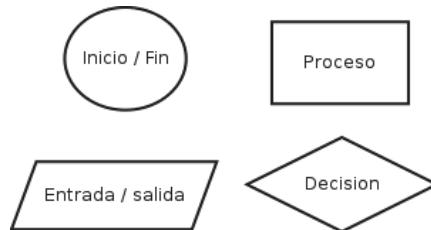
        Console.WriteLine("Introduce un número");
        numero = Convert.ToInt32(Console.ReadLine());
        if (numero = 0)
            Console.WriteLine("El número es cero.");
        else
            if (numero < 0)
                Console.WriteLine("El número es negativo.");
            else
        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

2.1.7. Introducción a los diagramas de flujo

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

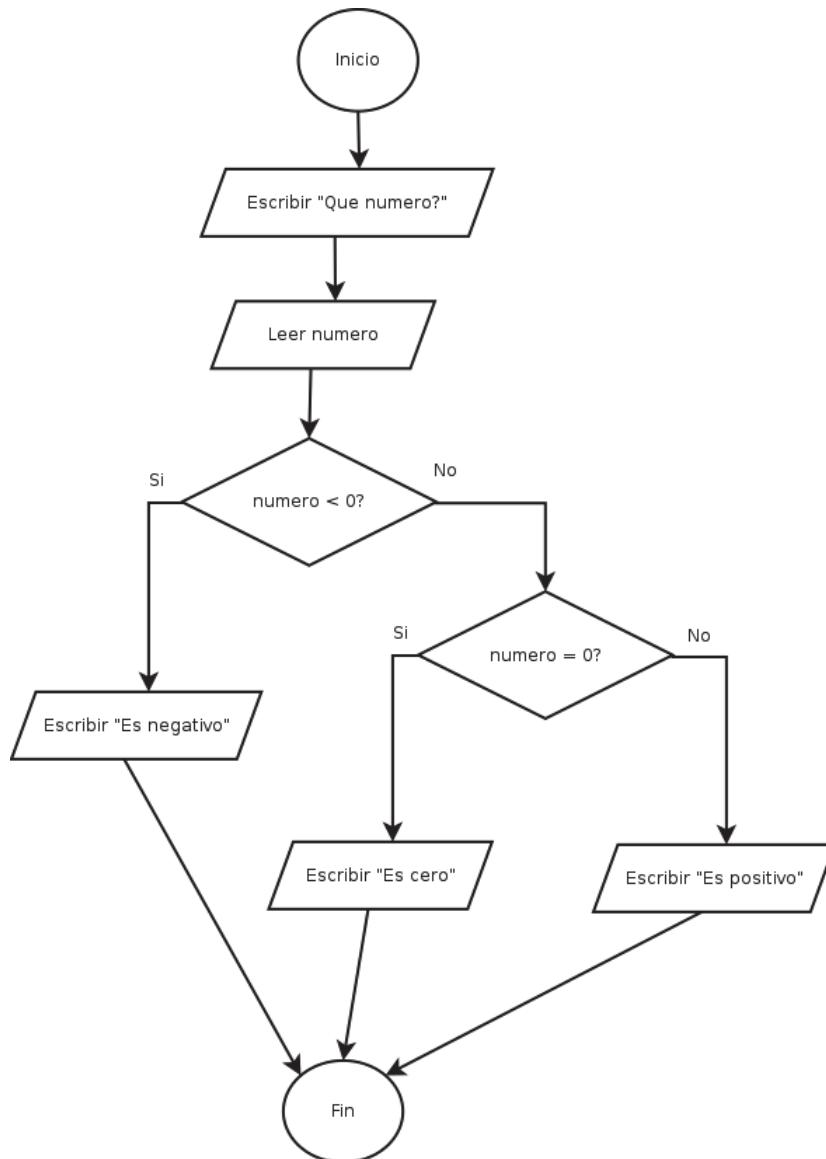
Para ayudarnos a centrarnos en el problema, existen notaciones gráficas, como los diagramas de flujo, que nos permiten ver mejor qué se debe hacer y cuando.

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



El inicio o el final del programa se indica dentro de un círculo. Los procesos internos, como realizar operaciones, se encuadran en un rectángulo. Las entradas y salidas (escrituras en pantalla y lecturas de teclado) se indican con un paralelogramo que tenga su lados superior e inferior horizontales, pero no tenga verticales los otros dos. Las decisiones se indican dentro de un rombo.

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



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

Ejercicios propuestos:

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

2.1.8. Operador condicional: ?

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

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

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

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

esto equivale a la siguiente orden "if":

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

Aplicado a un programa sencillo, podría ser

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

using System;

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

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

```

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

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

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

```

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

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

```

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

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

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

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

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

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

```

Ejercicios propuestos:

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

2.1.10. switch

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

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

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

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

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

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

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

        switch (letra)
        {
            case ' ': Console.WriteLine("Espacio.");
        }
    }
}
```

```

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

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

```

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

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

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

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

```

        case '8':
        case '9':
        case '0': Console.WriteLine("Dígito.");
                    break;
    default:   Console.WriteLine("Ni espacio ni dígito.");
                    break;
    }
}
}

```

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

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

```

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

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

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

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

```

Ejercicios propuestos:

- **(2.1.9.1)** Crear un programa que lea una letra tecleada por el usuario y diga si se trata de una vocal, una cifra numérica o una consonante (pista: habrá que usar un dato de tipo "char").
- **(2.1.9.2)** Crear un programa que lea una letra tecleada por el usuario y diga si se trata de un signo de puntuación, una cifra numérica o algún otro carácter.
- **(2.1.9.3)** Repetir el ejercicio 2.1.9.1, empleando "if" en lugar de "switch".
- **(2.1.9.4)** Repetir el ejercicio 2.1.9.2, empleando "if" en lugar de "switch".

2.2. Estructuras repetitivas

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

2.2.1. while

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

En el primer caso, su sintaxis es

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

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

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

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

using System;

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

        Console.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.

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

```

/*
 *-----*/
/* Ejemplo en C# nº 16b: */
/* ejemplo16b.cs */
/*
 *-----*/
/* Contar con "while" */
/*
 *-----*/
/* Introducción a C#, */
/* Nacho Cabanes */
/*-----*/


using System;

public class Ejemplo16b
{
    public static void Main()
    {
        int n = 1;

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

```

Ejercicios propuestos:

- **(2.2.1.1)** Crear un programa que pida al usuario su contraseña (numérica). Deberá terminar cuando introduzca como contraseña el número 1111, pero volvérsela a pedir tantas veces como sea necesario.
- **(2.2.1.2)** Crea un programa que escriba en pantalla los números del 1 al 10, usando "while".

- **(2.2.1.3)** Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "while".
- **(2.2.1.4)** Crear un programa calcule cuantas cifras tiene un número entero positivo (pista: se puede hacer dividiendo varias veces entre 10).
- **(2.2.1.5)** Crear el diagrama de flujo y la versión en C# de un programa que dé al usuario tres oportunidades para adivinar un número del 1 al 10.

2.2.2. do ... while

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

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

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

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

```
/*
 *-----*/
/* Ejemplo en C# nº 17: */
/* ejemplo17.cs */
/*
 *-----*/
/* La orden "do..while" */
/*
 *-----*/
/* Introduccion a C#, */
/* Nacho Cabanes */
/*-----*/
using System;

public class Ejemplo17
{
    public static void Main()
    {
        int valida = 711;
        int clave;

        do
        {

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

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

        }
        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 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":

```

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

public class Ejemplo18
{
    public static void Main()
    {

        string valida = "secreto";
        string clave;

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

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

        }
        while (clave != valida);

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

```

Ejercicios propuestos:

- **(2.2.2.1)** Crear un programa que pida números positivos al usuario, y vaya calculando la suma de todos ellos (terminará cuando se teclea un número negativo o cero).
- **(2.2.2.2)** Crea un programa que escriba en pantalla los números del 1 al 10, usando "do..while".

- **(2.2.2.3)** Crea un programa que escriba en pantalla los números pares del 26 al 10 (descendiendo), usando "do..while".
- **(2.2.2.4)** Crea un programa que pida al usuario su identificador y su contraseña (ambos numéricos), y no le permita seguir hasta que introduzca como identificador "1234" y como contraseña "1111".
- **(2.2.2.5)** Crea un programa que pida al usuario su identificador y su contraseña, y no le permita seguir hasta que introduzca como nombre "Pedro" y como contraseña "Peter".

2.2.3. for

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

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

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

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

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

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

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

using System;

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

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

```

    }
}
```

Ejercicios propuestos:

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

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

Por eso, si escribimos la siguiente línea:

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

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

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

```
for ( ; ; )
```

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

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

public class Ejemplo20
{
    public static void Main()
    {
        int tabla, numero;

        for (tabla=1; tabla<=5; tabla++)

```

```

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

```

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

```

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

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

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

            Console.WriteLine();
        }
    }
}

```

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

```

/*
/* Ejemplo en C# nº 22: */
/* ejemplo22.cs */

```

```

/*
/* "for" que usa "char"
*/
/* Introduccion a C#,
/* Nacho Cabanes
/*
-----*/
using System;

public class Ejemplo22
{
    public static void Main()
    {

        char letra;

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

    }
}

```

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

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

```

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

public class Ejemplo23
{
    public static void Main()
    {

        char letra;

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

    }
}

```

Ejercicios propuestos:

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

Nota: Se puede incluso declarar una nueva variable en el interior de "for", y esa variable desaparecerá cuando el "for" acabe:

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

2.3. Sentencia break: termina el bucle

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

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

using System;

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

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

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

El resultado de este programa es:

```
1 2 3 4
```

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

2.4. Sentencia continue: fuerza la siguiente iteración

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

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

using System;

public class Ejemplo25
{
    public static void Main()
    {

        int contador;

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

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

El resultado de este programa es:

1 2 3 4 6 7 8 9 10

En él podemos observar que no aparece el valor 5.

Ejercicios resueltos:

- ¿Qué escribiría en pantalla este fragmento de código?

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

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

- ¿Qué escribiría en pantalla este fragmento de código?

3. Tipos de datos básicos

3.1. Tipo de datos entero

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

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

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

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

3.1.1. Tipos de datos para números enteros

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

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

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

3.1.2. Conversiones de cadena a entero

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

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

Ejercicios propuestos:

- **(3.1.2.1)** Preguntar al usuario su edad, que se guardará en un "byte". A continuación, se deberá decir que no aparece tantos años (por ejemplo, "No apareces 20 años").
- **(3.1.2.2)** Pedir al usuario dos números de dos cifras ("byte"), calcular su multiplicación, que se deberá guardar en un "ushort", y mostrar el resultado en pantalla.
- **(3.1.2.3)** Pedir al usuario dos números enteros largos ("long") y mostrar 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, y que no tiene un símbolo específico para representarla en matemáticas: incrementar el valor de una variable en una unidad:

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

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

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

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

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

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

En cambio, si escribimos

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

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

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

Ejercicios propuestos:

- **(3.1.3.1)** Crear un programa que use tres variables x,y,z. Sus valores iniciales serán 15, -10, 2.147.483.647. Se deberá incrementar el valor de estas variables. ¿Qué valores esperas que se obtengan? Contrástalo con el resultado obtenido por el programa.
- **(3.1.3.2)** ¿Cuál sería el resultado de las siguientes operaciones? $a=5; b=++a;$
 $c=a++; b=b*5; a=a*2;$

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

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

3.1.4. Operaciones abreviadas: +=

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

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

Ejercicios propuestos:

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

3.2. Tipo de datos real

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

- **Coma fija:** el número máximo de cifras decimales está fijado de antemano, y el número de cifras enteras también. Por ejemplo, con un formato de 3 cifras enteras y 4 cifras decimales, el número 3,75 se almacenaría correctamente (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 1010 no se podría guardar (tiene más de 3 cifras enteras).
- **Coma flotante:** el número de decimales y de cifras enteras permitido es variable, lo que importa es el número de cifras significativas (a partir del último 0). Por ejemplo, con 5 cifras significativas se podrían almacenar números como el 13405000000 o como

el 0,0000007349 pero no se guardaría correctamente el 12,0000034, que se redondearía a un número cercano.

3.2.1. Simple y doble precisión

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

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

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

```
float x;
```

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

```
float x = 12.56;
```

3.2.2. Pedir y mostrar números reales

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

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

```
using System;

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

```

    float suma;

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

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

```

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

```

ejemplo05
Introduce el primer número
23,6
Introduce el segundo número
34,2
La suma de 23,6 y 342 es 365,6

```

Ejercicios propuestos:

- **(3.2.2.1)** Calcular el área de un círculo, dado su radio ($\pi * \text{radio al cuadrado}$)
- **(3.2.2.2)** Crear 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).
- **(3.2.2.3)** Hallar 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)`
- **(3.2.2.4)** 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: $\text{Resultado} = e^{(1+i)^n}$ Aplicarlo para calcular en cuanto se convierten 1.000 euros al cabo de 10 años al 3% de interés anual.
- **(3.2.2.5)** Crea un programa que muestre los primeros 20 valores de la función $y = x^2 - 1$
- **(3.2.2.6)** Crea un programa que "dibuje" la gráfica de $y = (x-5)^2$ para valores de x entre 1 y 10. Deberá hacerlo dibujando varios espacios en pantalla y luego un asterisco. La cantidad de espacios dependerá del valor obtenido para "y".
- **(3.2.2.7)** 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.

3.2.3. Formatear números

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

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

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

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

Vamos a probarlos en un ejemplo:

```
/*
 * Ejemplo en C# nº 28:
 */
/* ejemplo28.cs */
/*
 */
/* Formato de núms. reales */
/*
 */
/* Introduccion a C#,
 */
/* Nacho Cabanes */
/*
*/
using System;

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

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

El resultado de este ejemplo sería:

```
12,340
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:

- **(3.2.3.1)** 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.
- **(3.2.3.2)** Crear un programa que use tres variables x,y,z. Las tres serán números reales, y nos bastará con dos cifras decimales. Deberá pedir al usuario los valores para las tres variables y mostrar en pantalla el valor de $x^2 + y - z$ (con exactamente dos cifras decimales).
- **(3.2.3.3)** Calcular el perímetro, área y diagonal de un rectángulo, a partir de su ancho y alto (perímetro = suma de los cuatro lados, área = base x altura, diagonal usando el teorema de Pitágoras). Mostrar todos ellos con una cifra decimal.
- **(3.2.3.4)** Calcular la superficie y el volumen de una esfera, a partir de su radio (superficie = $4 * \pi * \text{radio}^2$; volumen = $4/3 * \pi * \text{radio}^3$). Mostrar los resultados con 3 cifras decimales.

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

```
using System;
public class Ejemplo28b
```

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

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

Su resultado sería:

```
f7
11110111
```

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

Ejercicios propuestos:

- **(3.2.4.1)** 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.
- **(3.2.4.2)** 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)
- **(3.2.4.3)** 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 de hexadecimal a binario o 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 = 0x12a3;" pero un valor precedido por "0" no se considera octal sino decimal, al contrario de lo que ocurre en los lenguajes C y C++:

```
/*
 *-----*
 * Ejemplo en C# nº 28c:      */
/* ejemplo28c.cs              */
/*                               */
/* Hexadecimal y binario       */
/* (2)                         */
/*                               */
/* Introduccion a C#,          */
/* Nacho Cabanes               */
/*-----*/
```

```
using System;

public class Example28c
{
    public static void Main()
    {
        int n1 = 0x12a3;
```

```

int n2 = Convert.ToInt32("12a4", 16);

int n3 = 0123; // No es octal, al contrario que en C y C++
int n4 = Convert.ToInt32("124", 8);

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

double d1 = 5.7;
float f1 = 5.7f;

Console.WriteLine( "{0} {1} {2} {3} {4}",
    n1, n2, n3, n4, n5);
Console.WriteLine( "{0} {1}",
    d1, f1);
}
}

```

Que mostraría:

```

4771 4772 123 84 201
5,7 5,7

```

Nota: La notación "float f1 = 5.7f;" se usa para detallar que se trata de un número real de simple precisión (un "float"), porque de lo contrario, en " float f1 = 5.7;" se consideraría un número de doble precisión, y al tratar de compilar obtendríamos un mensaje de error, diciendo que no se puede convertir de "double" a "float" sin pérdida de precisión. Al añadir la "f" al final, estamos diciendo "quiero que éste número se tome como un float; sé que habrá una pérdida de precisión pero es aceptable para mí".

Ejercicios propuestos:

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

3.3. Tipo de datos carácter

3.3.1. Leer y mostrar caracteres

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

```
char letra;
```

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

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

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

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

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

```
/*
 * Ejemplo en C# nº 29:
 * ejemplo29.cs
 *
 * Tipo de datos "char"
 *
 * Introduccion a C#,
 * Nacho Cabanes
 */

using System;

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

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

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

Ejercicio propuesto

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

3.3.2. Secuencias de escape: \n y otras

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

Estas secuencias especiales son las siguientes:

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

Vamos a ver un ejemplo que use los más habituales:

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

using System;

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

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

Su resultado sería este:

Esta es una frase

y esta es otra, separada dos lineas

Juguemos mas:

otro salto

Comillas dobles: " y simples ', y barra \

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

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

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

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

Ejercicio propuesto

- **(3.3.2.1)** Crea un programa que pida al usuario que teclee cuatro letras y las muestre en pantalla juntas, pero en orden inverso, y entre comillas dobles. Por ejemplo si las letras que se teclean son a, l, o, h, escribiría "hola".

3.4. Toma de contacto con las cadenas de texto

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

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

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

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

using System;

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

```

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

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

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

```

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

Ejercicios propuestos:

- **(3.4.1)** Crear un programa que pida al usuario su nombre, y le diga "Hola" si se llama "Juan", o bien le diga "No te conozco" si teclea otro nombre.
- **(3.4.2)** Crear un programa que pida al usuario un nombre y una contraseña. La contraseña se debe introducir dos veces. Si las dos contraseñas no son iguales, se avisará al usuario y se le volverán a pedir las dos contraseñas.

3.5. Los valores "booleanos"

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

```

bool encontrado;
encontrado = true;

```

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

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

```

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

```

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

```

/*-----*/
/* Ejemplo en C# nº 32:      */
/* ejemplo32.cs            */

```

```

/*
/* Condiciones con if (8) */
/* Variables bool */
/*
/* Introducción a C#, */
/* Nacho Cabanes */
/*-----*/
using System;

public class Ejemplo32
{
    public static void Main()
    {
        char letra;
        bool esVocal, esCifra;

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

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

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

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

```

Ejercicios propuestos:

- **(3.5.1)** Crear un programa que use el operador condicional para dar a una variable llamada "iguales" (booleana) el valor "true" si los dos números que ha tecleado el usuario son iguales, o "false" si son distintos.
- **(3.5.2)** Crea una versión alternativa del ejercicio 3.5.1, que use "if" en vez del operador condicional.
- **(3.5.3)** Crear un programa que use el operador condicional para dar a una variable llamada "ambosPares" (booleana) el valor "true" si dos números introducidos por el usuario son pares, o "false" si alguno es impar.
- **(3.5.4)** Crea una versión alternativa del ejercicio 3.5.3, que use "if" en vez del operador condicional.

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 intentar descomponer los problemas en trozos más pequeños, que sean más fáciles de resolver. Esto nos puede suponer varias ventajas:

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

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

5.2. Conceptos básicos sobre funciones

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

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

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

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

```
public static void Main()
{
    saludar();
    ...
}
```

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

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

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

using System;

public class Ejemplo47
{

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

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

Como ejemplo más detallado, la parte principal de una agenda o de una base de datos simple como las que hicimos en el tema anterior, podría ser simplemente:

```
leerDatosDeFichero();
do {
    mostrarMenu();
    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. No debe devolver ningún valor. 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. Crea también un "Main" que permita probarla.

- **(5.2.3)** Descompón en funciones la base de datos de ficheros (ejemplo 46), 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 además indicarle a nuestra función ciertos datos especiales con los que queremos que trabaje. Por ejemplo, si escribimos en pantalla números reales con frecuencia, nos puede resultar útil crear una función auxiliar que nos los muestre con el formato que nos interese. Lo podríamos hacer así:

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

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

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

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

El programa completo podría quedar así:

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

using System;

public class Ejemplo48
{

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

    public static void Main()
    {
        float x;
```

```

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

```

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

```

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

```

Ejercicios propuestos:

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

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

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

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

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

```

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

```

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

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

Un programa más detallado de ejemplo podría ser:

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

public class Ejemplo49
{
    public static int cuadrado ( int n )
    {
        return n*n;
    }

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

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

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

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

Ejercicios propuestos:

- **(5.4.1)** Crear una función que calcule el cubo de un número real (float) que se indique como parámetro. El resultado deberá ser otro número real. Probar esta función para calcular el cubo de 3.2 y el de 5.
- **(5.4.2)** Crear una función que calcule el menor de dos números enteros que recibirán como parámetros. El resultado será otro número entero.

- **(5.4.3)** Crear una función llamada "signo", que reciba un número real, y devuelva un número entero con el valor: -1 si el número es negativo, 1 si es positivo o 0 si es cero.
- **(5.4.4)** Crear una función que devuelva la primera letra de una cadena de texto. Probar esta función para calcular la primera letra de la frase "Hola".
- **(5.4.5)** Crear una función que devuelva la última letra de una cadena de texto. Probar esta función para calcular la última letra de la frase "Hola".
- **(5.4.6)** Crear una función que reciba un número y calcule y muestre en pantalla el valor del perímetro y de la superficie de un cuadrado que tenga como lado el número que se ha indicado como parámetro.

5.5. Variables locales y variables globales

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

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

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

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

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

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

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

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

```
using System;

public class Ejemplo50
{
```

```

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

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

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

public static void Main()
{
    int num1, num2;

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

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

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

```

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

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

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

Ejercicios propuestos:

- **(5.5.1)** Crear una función "pedirEntero", que reciba como parámetros el texto que se debe mostrar en pantalla, el valor mínimo aceptable y el valor máximo aceptable. Deberá pedir al usuario que introduzca el valor tantas veces como sea necesario, volvérsele a pedir en caso de error, y devolver un valor correcto. Probarlo con un programa que pida al usuario un año entre 1800 y 2100.
- **(5.5.2)** Crear una función "escribirTablaMultiplicar", que reciba como parámetro un número entero, y escriba la tabla de multiplicar de ese número (por ejemplo, para el 3 deberá llegar desde "3x0=0" hasta "3x10=30").
- **(5.5.3)** Crear 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)** Crear una función que reciba una cadena y una letra, y devuelva la cantidad de veces que dicha letra aparece en la cadena. Por ejemplo, si la cadena es "Barcelona" y la letra es 'a', debería devolver 2 (porque la "a" aparece 2 veces).
- **(5.5.5)** Crear una función que reciba un numero cualquiera y que devuelva como resultado la suma de sus dígitos. Por ejemplo, si el número fuera 123 la suma sería 6.
- **(5.5.6)** Crear una función que reciba una letra y un número, y escriba un "triángulo" formado por esa letra, que tenga como anchura inicial la que se ha indicado. Por ejemplo, si la letra es '*' y la anchura es 4, debería escribir

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

5.6. Los conflictos de nombres en las variables

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

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

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

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

El resultado de este programa es:

```
n vale 5
```

Ahora n vale 5

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

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

```
/*
 * Ejemplo en C# nº 52:
 * ejemplo52.cs
 *
 * Una variable global
 *
 * Introduccion a C#,
 * Nacho Cabanes
 */

using System;

public class Ejemplo52
{
    static int n = 7;

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

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

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

5.7. Modificando parámetros

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

```
/*
 * Ejemplo en C# nº 53:
 * ejemplo53.cs
 *
 * Modificar una variable
 * recibida como parámetro
 */
```

```

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

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

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

```

El resultado de este programa será:

```

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

```

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

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

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

```

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

```

```

using System;

public class Ejemplo54
{
    public static void duplica(ref int x) {
        Console.WriteLine(" El valor recibido vale {0}", x);
        x = x * 2;
        Console.WriteLine(" y ahora vale {0}", x);
    }

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

```

En este caso sí se modifica la variable n:

```

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

```

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

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

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

Ejercicios propuestos:

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

5.8. El orden no importa

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

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

using System;

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

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

5.9. Algunas funciones útiles

5.9.1. Números aleatorios

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

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

```
// Creamos un objeto random
Random generador = new Random();

// Generamos un número entre dos valores dados
```

```
// (el segundo límite no está incluido)
int aleatorio = generador.Next(1, 101);
```

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

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

Pero esta forma simplificada no sirve si necesitamos obtener dos números aleatorios a la vez, porque los dos se obtendrían en el mismo milisegundo y tendrían el mismo valor; en ese caso, usaríamos "Random" y llamaríamos dos veces a "Next".

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

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

using System;

public class Ejemplo56
{

    public static void Main()
    {

        Random r = new Random();
        int aleatorio = r.Next(1, 11);
        Console.WriteLine("Un número entre 1 y 10: {0}", aleatorio);
        int aleatorio2 = r.Next(1, 11);
        Console.WriteLine("Otro: {0}", aleatorio2);
    }
}
```

Ejercicios propuestos:

- **(5.9.1.1)** Crear un programa que genere un número al azar entre 1 y 100. El usuario tendrá 6 oportunidades para acertarlo.
- **(5.9.1.2)** Mejorar el programa del ahorcado (4.4.8.3), para que la palabra a adivinar no sea tecleado por un segundo usuario, sino que se escoja al azar de un "array" de palabras prefijadas (por ejemplo, nombres de ciudades).
- **(5.9.1.3)** Crea un programa que "dibuje" asteriscos en 100 posiciones al azar de la pantalla . Para ayudarte para escribir en cualquier coordenada, puedes usar un array de dos dimensiones (con tamaños 24 para el alto y 79 para el ancho), que primero rellenes y luego dibujes en pantalla.

5.9.2. Funciones matemáticas

En C# tenemos muchas funciones matemáticas predefinidas, como:

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

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

y una serie de constantes como

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

`PI`, el número "Pi", 3.14159...

Todas ellas se usan precedidas por "Math."

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

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

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

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

Ejercicios propuestos:

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

13. Otras características avanzadas de C#

13.1. Espacios de nombres

Desde nuestros primeros programas hemos estado usando cosas como "System.Console" o bien "using System". Esa palabra "System" indica que las funciones que estamos usando pertenecen a la estructura básica de C# y de la plataforma .Net.

La idea detrás de ese "using" es que puede ocurrir que distintos programadores en distintos puntos del mundo creen funciones o clases que se llamen igual, y, si se mezclan fuentes de distintas procedencias, esto podría dar lugar a programas que no compilaran correctamente, o, peor aún, que compilaran pero no funcionaran de la forma esperada.

Por eso, se recomienda usar "espacios de nombres", que permitan distinguir unos de otros. Por ejemplo, si yo quisiera crear mi propia clase "Console" para el acceso a la consola, o mi propia clase "Random" para manejo de números aleatorios, lo razonable es crear un nuevo espacio de nombres.

De hecho, con entornos como SharpDevelop o Visual Studio, cuando creamos un nuevo proyecto, el fuente "casi vacío" que se nos propone contendrá un espacio de nombres que se llamará igual que el proyecto. Esta es apariencia del fuente si usamos VisualStudio 2008:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Y esta es apariencia del fuente si usamos SharpDevelop 3:

```
using System;

namespace PruebaDeNamespaces
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            // TODO: Implement Functionality Here
            Console.Write("Press any key to continue . . . ");
        }
    }
}
```

```
        Console.ReadKey(true);  
    }  
}  
}
```

Vamos a un ejemplo algo más avanzado, que contenga un espacio de nombres, que cree una nueva clase Console y que utilice las dos (la nuestra y la original, de System):

```
/*-----*/  
/* Ejemplo en C# */  
/* namespaces.cs */  
/*-----*/  
/* Ejemplo de espacios de */  
/*   nombres */  
/*-----*/  
/* Introduccion a C#, */  
/*   Nacho Cabanes */  
/*-----*/  
  
using System;  
  
namespace ConsolaDeNacho {  
    public class Console  
    {  
        public static void WriteLine(string texto)  
        {  
            System.Console.ForegroundColor = ConsoleColor.Blue;  
            System.Console.WriteLine("Mensaje: "+texto);  
        }  
    }  
}  
  
public class PruebaDeNamespaces  
{  
  
    public static void Main()  
    {  
        System.Console.WriteLine("Hola");  
        ConsolaDeNacho.Console.WriteLine("Hola otra vez");  
    }  
}
```

Como se puede ver, este ejemplo tiene dos clases Console, y ambas tienen un método WriteLine. Una es la original de C#, que invocaríamos con "System.Console". Otra es la que hemos creado para el ejemplo, que escribe un texto modifica y en color (ayudándose de System.Console), y que llamaríamos mediante "ConsolaDeNacho.Console". El resultado es que podemos tener dos clases Console accesibles desde el mismo programa, sin que existan conflictos entre ellas. El resultado del programa sería:

```
Hola  
Mensaje: Hola otra vez
```

6. Programación orientada a objetos

6.1. ¿Por qué los objetos?

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

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

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

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

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

Con esto, hemos comentado casi sin saberlo las tres características más importantes de la Programación Orientada a Objetos (OOP):

- **Encapsulación:** No podemos separar los comportamientos de las características de un objeto. Los comportamientos serán funciones, que en OOP llamaremos **métodos**. Las características de un objeto serán variables, como las que hemos usado siempre (las llamaremos **atributos**). La apariencia de un objeto en C#, como veremos un poco más adelante, recordará a un registro o "struct".
- **Herencia:** Unos objetos pueden heredar métodos y datos de otros. Esto hace más fácil definir objetos nuevos a partir de otros que ya teníamos anteriormente (como ocurría con el portón y la puerta) y facilitará la reescritura de los programas, pudiendo aprovechar buena parte de los anteriores... si están bien diseñados.
- **Polimorfismo:** Un mismo nombre de un método puede hacer referencia a comportamientos distintos (como abrir una puerta o un libro). Igual ocurre para los datos: el peso de una puerta y el de un portón los podemos llamar de igual forma, pero obviamente no valdrán lo mismo.

Otro concepto importante es el de "clase": **Una clase** es un conjunto de objetos que tienen características comunes. Por ejemplo, tanto mi puerta como la de mi vecino son puertas, es decir, ambas son objetos que pertenecen a la clase "puerta". De igual modo, tanto un Ford Focus como un Honda Civic o un Toyota Corolla son objetos concretos que pertenecen a la clase "coche".

6.2. Objetos y clases en C#

Vamos con los detalles. Las **clases en C#** se definen de forma parecida a los registros (struct), sólo que ahora también incluirán funciones. Así, la clase "Puerta" que mencionábamos antes se podría declarar así:

```
public class Puerta
{
    int ancho;      // Ancho en centimetros
    int alto;       // Alto en centimetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
} // Final de la clase Puerta
```

Como se puede observar, los objetos de la clase "Puerta" tendrán un ancho, un alto, un color, y un estado (abierta o no abierta), y además se podrán abrir o cerrar (y además, nos pueden "mostrar su estado, para comprobar que todo funciona correctamente").

Para declarar estos objetos que pertenecen a la clase "Puerta", usaremos la palabra "new", igual que hacíamos con los "arrays":

```
Puerta p = new Puerta();
p.Abrir();
p.MostrarEstado();
```

Vamos a completar un programa de prueba que use un objeto de esta clase (una "Puerta"):

```
/*
 * Ejemplo en C# nº 59:      */
/* ejemplo59.cs               */
/*                               */
/* Primer ejemplo de clases   */
/*                               */
/* Introducción a C#,          */
/* Nacho Cabanes               */
/*-----*/
using System;

public class Puerta
{
    int ancho;      // Ancho en centímetros
    int alto;       // Alto en centímetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
} // Final de la clase Puerta

public class Ejemplo59
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine("\nVamos a abrir...");
        p.Abrir();
        p.MostrarEstado();
    }
}
```

Este fuente ya no contiene una única clase (class), como todos nuestros ejemplos anteriores, sino dos clases distintas:

- La clase "Puerta", que son los nuevos objetos con los que vamos a practicar.
- La clase "Ejemplo59", que representa a nuestra aplicación.

El resultado de ese programa es el siguiente:

Valores iniciales...

Ancho: 0

Alto: 0

Color: 0

Abierta: False

Vamos a abrir...

Ancho: 0

Alto: 0

Color: 0

Abierta: True

Se puede ver que en C#, al contrario que en otros lenguajes, las variables que forman parte de una clase (los "atributos") tienen un valor inicial predefinido: 0 para los números, una cadena vacía para las cadenas de texto, o "False" para los datos booleanos.

Vemos también que se accede a los métodos y a los datos precediendo el nombre de cada uno por el nombre de la variable y por un punto, como hacíamos con los registros (struct). Aun así, en nuestro caso no podemos hacer directamente "p.abierta = true", por dos motivos:

- El atributo "abierta" no tiene delante la palabra "public"; por lo que no es público, sino privado, y no será accesible desde otras clases (en nuestro caso, desde Ejemplo59).
- Los puristas de la Programación Orientada a Objetos recomiendan que no se acceda directamente a los atributos, sino que siempre se modifiquen usando métodos auxiliares (por ejemplo, nuestro "Abrir"), y que se lea su valor también usando una función. Esto es lo que se conoce como "**ocultación de datos**". Supondrá ventajas como que podremos cambiar los detalles internos de nuestra clase sin que afecte a su uso.

Normalmente, como forma de ocultar datos, crearemos funciones auxiliares GetXXX y SetXXX que permitan acceder al valor de los atributos (en C# existe una forma alternativa de hacerlo, usando "propiedades", que veremos más adelante):

```
public int GetAncho()
{
    return ancho;
}

public void SetAncho(int nuevoValor)
{
    ancho = nuevoValor;
```

}

También puede desconcertar que en "Main" aparezca la palabra "**static**", mientras que no lo hace en los métodos de la clase "Puerta". Veremos este detalle un poco más adelante, en el tema 7, pero de momento perderemos la costumbre de escribir "static" antes de cada función: a partir de ahora, sólo Main lo será.

Ejercicio propuesto:

- **(6.2.1)** Crear una clase llamada Persona, en el fichero "persona.cs". Esta clase deberá tener un atributo "nombre", de tipo string. También deberá tener un método "SetNombre", de tipo void y con un parámetro string, que permita cambiar el valor del nombre. Finalmente, también tendrá un método "Saludar", que escribirá en pantalla "Hola, soy " seguido de su nombre. Crear también una clase llamada PruebaPersona. Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre y les pedirá que saluden.

En un proyecto grande, es recomendable que cada clase esté en su propio fichero fuente, de forma que se puedan localizar con rapidez (en los que hemos hecho en el curso hasta ahora, no era necesario, porque eran muy simples). Precisamente por eso, es interesante (pero no obligatorio) que cada clase esté en un fichero que tenga el mismo nombre: que la clase Puerta se encuentre en el fichero "Puerta.cs". Esta es una regla que no seguiremos en algunos de los ejemplos del texto, por respetar la numeración consecutiva de los ejemplos, pero que sí se debería seguir en un proyecto de mayor tamaño, formado por varias clases.

Para **compilar un programa formado por varios fuentes**, basta con indicar los nombres de todos ellos. Por ejemplo, con Mono sería

```
gmcs fuente1.cs fuente2.cs fuente3.cs
```

En ese caso, el ejecutable obtenido tendría el nombre del primero de los fuentes (fuente1.exe). Podemos cambiar el nombre del ejecutable con la opción "-out" de Mono:

```
gmcs fuente1.cs fuente2.cs fuente3.cs -out:ejemplo.exe
```

(Un poco más adelante veremos cómo crear un programa formado por varios fuentes usando Visual Studio y otros entornos integrados).

Vamos a dividir en dos fuentes el último ejemplo y a ver cómo se compilaría. La primera clase podría ser ésta:

```
/*-----*/
/* Ejemplo en C# nº 59b: */
/* ejemplo59b.cs */
/*
/* Dos clases en dos */
/* ficheros (fichero 1) */
```

```

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

public class Puerta
{

    int ancho;      // Ancho en centimetros
    int alto;       // Alto en centimetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
} // Final de la clase Puerta

```

Y la segunda clase podría ser:

```

/*
 * Ejemplo en C# nº 59c:      */
/* ejemplo59c.cs            */
/*                               */
/* Dos clases en dos         */
/* ficheros (fichero 2)       */
/*                               */
/* Introduccion a C#,        */
/* Nacho Cabanes             */
/*-----*/
public class Ejemplo59c
{
    public static void Main()
    {
        Puerta p = new Puerta();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();
    }
}

```

```

        Console.WriteLine("\nVamos a abrir...");  

        p.Abrir();  

        p.MostrarEstado();  

    }  

}

```

Y lo compilaríamos con:

```
gmcs ejemplo59b.cs ejemplo59c.cs -out:ejemplo59byc.exe
```

Aun así, para estos proyectos formados por varias clases, lo ideal es usar algún entorno más avanzado, como SharpDevelop o VisualStudio, que permitan crear todas las clases con comodidad, saltar de una clase a clase otra rápidamente, que marquen dentro del propio editor la línea en la que están los errores... por eso, al final de este tema tendrás un apartado con una introducción al uso de SharpDevelop.

Ejercicio propuesto:

- **(6.2.2)** Modificar el fuente del ejercicio anterior, para dividirlo en dos ficheros: Crear una clase llamada Persona, en el fichero "persona.cs". Esta clase deberá tener un atributo "nombre", de tipo string. También deberá tener un método "SetNombre", de tipo void y con un parámetro string, que permita cambiar el valor del nombre. Finalmente, también tendrá un método "Saludar", que escribirá en pantalla "Hola, soy " seguido de su nombre. Crear también una clase llamada PruebaPersona, en el fichero "pruebaPersona.cs". Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre y les pedirá que saluden.

6.3. La herencia. Visibilidad

Vamos a ver ahora cómo definir una nueva clase de objetos a partir de otra ya existente. Por ejemplo, vamos a crear una clase "Porton" a partir de la clase "Puerta". Un portón tendrá las mismas características que una puerta (ancho, alto, color, abierto o no), pero además se podrá bloquear, lo que supondrá un nuevo atributo y nuevos métodos para bloquear y desbloquear:

```

public class Porton: Puerta  

{  

    bool bloqueada;  

    public void Bloquear()  

    {  

        bloqueada = true;  

    }  

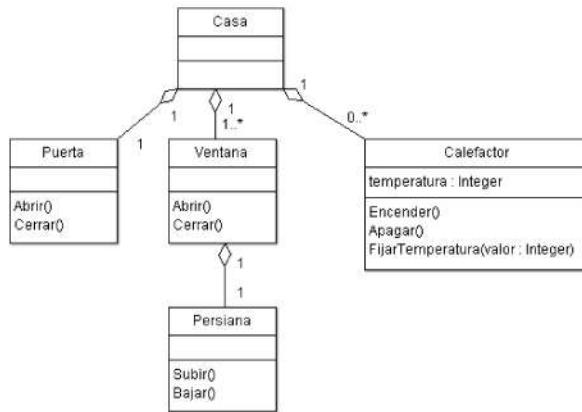
  

    public void Desbloquear()  

    {  

        bloqueada = false;
    }
}

```



Este diagrama es una forma más simple de ver las clases existentes y las relaciones entre ellas. Si generamos las clases a partir del diagrama, tendremos parte del trabajo hecho: ya "sólo" nos quedará rellenar los detalles de métodos como "Abrir", pero el esqueleto de todas las clases ya estará "escrito" para nosotros.

Ejercicios propuestos:

- **(6.4.1)** Crear las clases correspondientes al diagrama de clases del apartado 6.4, con sus correspondientes métodos y atributos.

6.5. Constructores y destructores.

Hemos visto que al declarar una clase, se dan valores por defecto para los atributos. Por ejemplo, para un número entero, se le da el valor 0. Pero puede ocurrir que nosotros deseemos dar valores iniciales que no sean cero. Esto se puede conseguir declarando un "**constructor**" para la clase.

Un **constructor** es una función especial, que se pone en marcha cuando se crea un objeto de una clase, y se suele usar para dar esos valores iniciales, para reservar memoria si fuera necesario, etc.

Se declara usando el mismo nombre que el de la clase, y sin ningún tipo de retorno. Por ejemplo, un "constructor" para la clase **Puerta** que le diera los valores iniciales de 100 para el ancho, 200 para el alto, etc., podría ser así:

```

public Puerta()
{
    ancho = 100;
    alto = 200;
    color = 0xFFFFFFFF;
    abierta = false;
}
  
```

Podemos tener más de un constructor, cada uno con distintos parámetros. Por ejemplo, puede haber otro constructor que nos permita indicar el ancho y el alto:

```
public Puerta(int an, int al)
{
    ancho = an;
    alto = al;
    color = 0xFFFFFFFF;
    abierta = false;
}
```

Ahora, si declaramos un objeto de la clase puerta con "Puerta p = new Puerta();" tendrá de ancho 100 y de alto 200, mientras que si lo declaramos con "Puerta p2 = new Puerta(90,220);" tendrá 90 como ancho y 220 como alto.

Un programa de ejemplo que usara estos dos constructores para crear dos puertas con características iniciales distintas podría ser:

```
/*
 * Ejemplo en C# nº 61:
 * ejemplo61.cs
 *
 * Tercer ejemplo de clases
 * Constructores
 *
 * Introducción a C#,
 * Nacho Cabanes
 */

using System;

public class Puerta
{

    int ancho;      // Ancho en centímetros
    int alto;       // Alto en centímetros
    int color;      // Color en formato RGB
    bool abierta;   // Abierta o cerrada

    public Puerta()
    {
        ancho = 100;
        alto = 200;
        color = 0xFFFFFFFF;
        abierta = false;
    }

    public Puerta(int an, int al)
    {
        ancho = an;
        alto = al;
        color = 0xFFFFFFFF;
        abierta = false;
    }
}
```

```

public void Abrir()
{
    abierta = true;
}

public void Cerrar()
{
    abierta = false;
}

public void MostrarEstado()
{
    Console.WriteLine("Ancho: {0}", ancho);
    Console.WriteLine("Alto: {0}", alto);
    Console.WriteLine("Color: {0}", color);
    Console.WriteLine("Abierta: {0}", abierta);
}

} // Final de la clase Puerta

public class Ejemplo61
{

    public static void Main()
    {
        Puerta p = new Puerta();
        Puerta p2 = new Puerta(90,220);

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine("\nVamos a abrir...");
        p.Abrir();
        p.MostrarEstado();

        Console.WriteLine("Para la segunda puerta...");
        p2.MostrarEstado();
    }
}

```

Nota: al igual que existen los "constructores", también podemos indicar un "**destructor**" para una clase, que se encargue de liberar la memoria que pudiéramos haber reservado en nuestra clase (no es nuestro caso, porque aún no sabemos manejar memoria dinámica) o para cerrar ficheros abiertos (que tampoco sabemos).

Un "**destructor**" se llama igual que la clase, pero precedido por el símbolo "~", no tiene tipo de retorno, y no necesita ser "public", como ocurre en este ejemplo:

```

~Puerta()
{
    // Liberar memoria
    // Cerrar ficheros
}

```

7. Utilización avanzada de clases

7.1. La palabra "static"

Desde un principio, nos hemos encontrado con que "Main" siempre iba acompañado de la palabra "static". En cambio, los métodos (funciones) que pertenecen a nuestros objetos no los estamos declarando como "static". Vamos a ver el motivo:

La palabra "static" delante de un atributo (una variable) de una clase, indica que es una "variable de clase", es decir, que su valor es el mismo para todos los objetos de la clase. Por ejemplo, si hablamos de coches convencionales, podríamos suponer que el atributo "numeroDeRuedas" va a valer 4 para cualquier objeto que pertenezca a esa clase (cualquier coche). Por eso, se podría declarar como "static".

De igual modo, si un método (una función) está precedido por la palabra "static", indica que es un "método de clase", es decir, un método que se podría usar sin necesidad de declarar ningún objeto de la clase. Por ejemplo, si queremos que se pueda usar la función "BorrarPantalla" de una clase "Hardware" sin necesidad de crear primero un objeto perteneciente a esa clase, lo podríamos conseguir así:

```
public class Hardware
{
    ...
    public static void BorrarPantalla ()
    {
        ...
    }
}
```

que desde dentro de "Main" (incluso perteneciente a otra clase) se usaría con el nombre de la clase delante:

```
public class Juego
{
    ...
    public ComienzoPartida()
    {
        Hardware.BorrarPantalla ();
        ...
    }
}
```

Desde una función "static" no se puede llamar a otras funciones que no lo sean. Por eso, como nuestro "Main" debe ser static, deberemos siempre elegir entre:

- Que todas las demás funciones de nuestro fuente también estén declaradas como "static", por lo que podrán ser utilizadas desde "Main" (como hicimos en el tema 5).
- Declarar un objeto de la clase correspondiente, y entonces sí podremos acceder a sus métodos desde "Main":

- **(7.4.2)** Crea una versión ampliada del ejercicio 7.4.1, en la que el constructor de todas las clases hijas se apoye en el de la clase "Trabajador".

7.5. La palabra "this": el objeto actual

Podemos hacer referencia al objeto que estamos usando, con "this":

```
public void MoverA (int x, int y)
{
    this.x = x;
    this.y = y;
}
```

En muchos casos, podemos evitarlo. Por ejemplo, normalmente es preferible usar otro nombre en los parámetros:

```
public void MoverA (int nuevaX, int nuevaY)
{
    this.x = nuevaX;
    this.y = nuevaY;
}
```

Y en ese uso se puede (y se suele) omitir "this":

```
public void MoverA (int nuevaX, int nuevaY)
{
    x = nuevaX;
    y = nuevaY;
}
```

Pero "this" tiene también otros usos. Por ejemplo, podemos crear un **constructor** a partir de otro que tenga distintos parámetros:

```
public RectanguloRelleno (int x, int y )
    : this (x)
{
    fila = y;
    // Pasos adicionales
}
```

Y se usa mucho para que unos objetos "conozcan" a los otros:

```
public void IndicarEnemigo (ElemGrafico enemigo)
{
    ...
}
```

De modo que el personaje de un juego le podría indicar al adversario que él es su enemigo con

```
miAdversario.IndicarEnemigo(this);
```

Es habitual usarlo para que una clase sepa a cuál pertenece (por ejemplo, a qué casa pertenece una puerta), porque de lo contrario, cada clase "conoce" a los objetos que contiene (la casa "conoce" a sus puertas), pero no al contrario.

Ejercicios propuestos:

- **(7.5.1)** Crea una versión ampliada del ejercicio 7.4.2, en la que el constructor sin parámetros de la clase "Trabajador" se apoye en otro constructor que reciba como parámetro el nombre de esa persona. La versión sin parámetros asignará el valor "Nombre no detallado" al nombre de esa persona.

7.6. Sobrecarga de operadores

Los "operadores" son los símbolos que se emplean para indicar ciertas operaciones. Por ejemplo, el operador "+" se usa para indicar que queremos sumar dos números.

Pues bien, en C# se puede "sobrecargar" operadores, es decir, redefinir su significado, para poder sumar (por ejemplo) objetos que nosotros hayamos creado, de forma más cómoda y legible. Por ejemplo, para sumar dos matrices, en vez de hacer algo como "matriz3 = suma(matriz1, matriz2)" podríamos hacer simplemente " matriz3 = matriz1 + matriz2"

No entraremos en detalle, pero la idea está en que redefiniríamos un método llamado "operador +", y que devolvería un dato del tipo con el que estamos (por ejemplo, una Matriz) y recibiría dos datos de ese mismo tipo como parámetros:

```
public static Matriz operator +(Matriz mat1, Matriz mat2)
{
    Matriz nuevaMatriz = new Matriz();

    for (int x=0; x < tamanyo; x++)
        for (int y=0; y < tamanyo; y++)
            nuevaMatriz[x, y] = mat1[x, y] + mat2[x, y];

    return nuevaMatriz;
}
```

Desde "Main", calcularíamos una matriz como suma de otras dos haciendo simplemente

```
Matriz matriz3 = matriz1 + matriz2;
```

Ejercicios propuestos:

- **(7.6.1)** Desarrolla una clase "Matriz", que represente a una matriz de 3x3, con métodos para indicar el valor que hay en una posición, leer el valor de una posición, escribir la matriz en pantalla y sumar dos matrices.

4. Arrays, estructuras y cadenas de texto

4.1. Conceptos básicos sobre arrays o tablas

4.1.1. Definición de un array y acceso a los datos

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

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

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

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

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

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

public class Ejemplo33
{
    public static void Main()
    {

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

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

}

Ejercicios propuestos:

- **(4.1.1.1)** Un programa que pida al usuario 4 números, los memorice (utilizando una tabla), calcule su media aritmética y después muestre en pantalla la media y los datos tecleados.
- **(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.
- **(4.1.1.3)** 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 cada uno de los valores que tiene almacenados con el que hasta ese momento es el máximo. El valor inicial de este máximo no debería ser cero (porque fallaría si todos los números son negativos), sino el primer elemento del array.

4.1.2. Valor inicial de un array

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

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

using System;

public class Ejemplo34
{
    public static void Main()
    {

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

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

Ejercicios propuestos:

- **(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 = numero[0] + numero[1] + numero[2] + numero[3] + numero[4];
```

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

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

El fuente completo podría ser así:

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

using System;

public class Ejemplo35
{
    public static void Main()
    {

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

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

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

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

Ejercicios propuestos:

- **(4.1.3.1)** 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).

4.1.4. Datos repetitivos introducidos por el usuario

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

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

public class Ejemplo36
{
    public static void Main()
    {

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

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

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

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

Ejercicios propuestos:

- **(4.1.4.1)** A partir del ejercicio 4.1.3.1, crear otro que pida al usuario que le indique la fecha, detallando el día (1 al 31) y el mes (1=enero, 12=diciembre), como respuesta muestre en pantalla el número de días que quedan hasta final de año.
- **(4.1.4.2)** Crear un programa que pida al usuario 10 números en coma flotante (pista: necesitarás un array de "float") y luego los muestre en orden inverso (del último que se ha introducido al primero que se introdujo).
- **(4.1.4.3)** Un programa que pida al usuario 10 números y luego calcule y muestre cuál es el mayor de todos ellos.
- **(4.1.4.4)** Un programa que pida al usuario 10 números, calcule su media y luego muestre los que están por encima de la media.
- **(4.1.4.5)** Un programa que pida 10 nombres y los memorice (pista: esta vez se trata de un array de "string"). Después deberá pedir que se teclee un nombre y dirá si se encuentra o no entre los 10 que se han tecleado antes. Volverá a pedir otro nombre y a decir si se encuentra entre ellos, y así sucesivamente hasta que se teclee "fin".
- **(4.1.4.6)** Un programa que准备 espacio para un máximo de 100 nombres. El usuario deberá ir introduciendo un nombre cada vez, hasta que se pulse Intro sin teclear nada, momento en el que dejarán de pedirse más nombres y se mostrará en pantalla la lista de los nombres que se han introducido.
- **(4.1.4.7)** Un programa que reserve espacio para un vector de 3 componentes, pida al usuario valores para dichas componentes (por ejemplo [2, -5, 7]) y muestre su módulo (raíz cuadrada de la suma de sus componentes al cuadrado).
- **(4.1.4.8)** 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).
- **(4.1.4.9)** Un programa que reserve espacio para dos vectores de 3 componentes, pida al usuario sus valores y calcule su producto escalar ($x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$).

4.1.5. Operaciones habituales: buscar, añadir, insertar, borrar

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

- Para ver si un dato existe, habrá que recorrer todo el array, comparando el valor almacenado con el dato que se busca. Puede interesarnos simplemente saber si está o no (con lo que se podría interrumpir la búsqueda en cuanto aparezca una primera vez) o ver en qué posiciones se encuentra (para lo que habría que recorrer todo el array). Si el array estuviera ordenado, se podría buscar de una forma más rápida, pero la veremos más adelante.
- Para 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, para poder guardar el dato en la primera posición libre.
- Para insertar un dato en una cierta posición, los que quedén detrás deberán desplazarse "a la derecha" para dejarle hueco. Este movimiento debe empezar desde el final para que cada dato que se mueve no destruya el que estaba a continuación de él.

También habrá que actualizar el contador, para indicar que queda una posición libre menos.

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

Vamos a verlo con un ejemplo:

```
/*
 * Ejemplo en C# nº 36b: ejemplos36b.cs
 */
/*
 * Añadir y borrar en un array
 */
/*
 * Introducción a C#, Nacho Cabanes
 */
using System;

public class Ejemplo36b
{
    public static void Main()
    {

        int[] dato = {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} ",dato[i]);
        Console.WriteLine();

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

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

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

```

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

// Y volvemos a mostrar el array
for (i=0; i<cantidad; i++)
    Console.Write("{0} ",dato[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--)
        dato[i] = dato[i-1];
    dato[posicionInsertar] = 30;
    cantidad++;
}

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

```

que tendría como resultado:

```

10 15 12
15 encontrado en la posición 2
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 (no sirve emplear un "else", porque en cada pasada del bucle "for" no sabemos si el dato no existe, sólo que no está en la posición actual).

Ejercicios propuestos:

- **(4.1.5.1)** Amplía el ejemplo anterior (36b) para que avise si el dato buscado no aparece.
- **(4.1.5.2)** 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 quedén detrás deberán desplazarse "a la derecha" para dejarle hueco; por

- ejemplo, si el array contiene "hola", "adios" y se pide insertar "bien" en la segunda posición, el array pasará a contener "hola", "bien", "adios".
- Borrar el dato que hay en una cierta posición (como se ha visto, lo que estaban detrás deberán desplazarse "a la izquierda" para que no haya huecos; por ejemplo, si el array contiene "hola", "bien", "adios" y se pide borrar el dato de la segunda posición, el array pasará a contener "hola", "adios"
 - Mostrar los datos que contiene el array.
 - Salir del programa.

4.2. Tablas bidimensionales

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

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

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

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

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

```
using System;

public class Ejemplo37
{
    public static void Main()
```

```

    {

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

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

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

```

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

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

```

/*
 *----- Ejemplo en C# nº 38: -----
 *----- ejemplo38.cs -----
 *----- -----
 *----- Array de dos dimensiones -----
 *----- al estilo C... o casi -----
 *----- -----
 *----- Introduccion a C#,
 *----- Nacho Cabanes -----
 *----- -----
 */

using System;

public class Ejemplo38
{
    public static void Main()
    {

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

        // Damos valores de ejemplo
        for (int i=0;i<notas.Length;i++)

```

```

{
    for (int j=0;j<notas[i].Length;j++)
    {
        notas[i][j] = i + j;
    }
}

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

```

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.
- Si has estudiado álgebra de matrices:
 - **(4.2.3)** Un programa que calcule el determinante de una matriz de 2x2.
 - **(4.2.4)** Un programa que calcule el determinante de una matriz de 3x3.
 - **(4.2.5)** Un programa que calcule si las filas de una matriz son linealmente dependientes.
 - **(4.2.6)** Un programa que use matrices para resolver un sistema de ecuaciones lineales mediante el método de Gauss.

4.3. Estructuras o registros

4.3.1. Definición y acceso a los datos

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

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

```

public static void Main()
{
    tipoPersona persona;

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

```

Ejercicios propuestos:

- **(4.3.3.1)** Ampliar el programa 4.3.2.1, para que el campo "duración" se almacene como minutos y segundos, usando un "struct" anidado que contenga a su vez estos dos campos.

4.4. Cadenas de caracteres

4.4.1. Definición. Lectura desde teclado

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

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

```

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

public class Ejemplo42
{

    public static void Main()
    {

        string saludo = "Hola";
        string segundoSaludo;
        string nombre, despedida;

```

```

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

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

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

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

```

4.4.2. Cómo acceder a las letras que forman una cadena

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

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

4.4.3. Longitud de la cadena.

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

Ejercicios propuestos:

- **(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 muestra en orden inverso (de la última letra a la primera).
- **(4.4.3.3)** Un programa capaz de sumar dos números enteros muy grandes (por ejemplo, de 30 cifras), que se deberán pedir como cadena de texto y analizar letra a letra.
- **(4.4.3.4)** Un programa capaz de multiplicar dos números enteros muy grandes (por ejemplo, de 30 cifras), que se deberán pedir como cadena de texto y analizar letra a letra.

4.4.4. Extraer una subcadena

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

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

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

Ejercicios propuestos:

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

```
J  
Ju  
Jua  
Juan
```

4.4.5. Buscar en una cadena

Para ver si una cadena contiene un cierto texto, podemos usar `IndexOf ("posición de")`, que nos dice en qué posición se encuentra, siendo 0 la primera posición (o devuelve el valor -1, si no aparece):

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

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 el que, en caso de que alguno de los textos aparezca como subcadena, se avise además si se encuentra exactamente al principio.

4.4.6. Otras manipulaciones de cadenas

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

- ToUpper() convierte a mayúsculas: nombreCorrecto = nombre.ToUpper();
- ToLower() convierte a minúsculas: password2 = password.ToLower();
- Insert(int posición, string subcadena): Insertar una subcadena en una cierta posición de la cadena inicial: nombreFormal = nombre.Insert(0,"Don");
- Remove(int posición, int cantidad): Elimina una cantidad de caracteres en cierta posición: apellidos = nombreCompleto.Remove(0,6);
- Replace(string textoASustituir, string cadenaSustituta): Sustituye una cadena (todas las veces que aparezca) por otra: nombreCorregido = nombre.Replace("Pepe", "Jose");

Un programa que probara todas estas posibilidades podría ser así:

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

using System;

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

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

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

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

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

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

        Console.WriteLine("La ultima A esta en la posicion: {0}",
            
```

```

        ejemplo.LastIndexOf("a"));

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

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

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

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

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

}

```

Y su resultado sería

```

El texto es: Hola, que tal estas
La primera letra es: H
Las tres primeras letras son: Hol
La longitud del texto es: 19
La 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 ", tio": Hola, tio, que tal estas
Si borramos las 6 primeras letras: que tal estas
Si cambiamos ESTAS por ESTAMOS: Hola, que tal estamos

```

Ejercicios propuestos:

- **(4.4.6.1)** Una variante del ejercicio 4.4.5.2, que no distinga entre mayúsculas y minúsculas a la hora de buscar.
- **(4.4.6.2)** Un programa que pida al usuario una frase y elimine todos los espacios redundantes que contenga (debe quedar sólo un espacio entre cada palabra y la siguiente).

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

```

/*
-----*/
/* Ejemplo en C# nº 43b: */
/* ejemplo43b.cs */
/*

```

```

/*
 * Cadenas de texto (2b)
 */
/*
 * Introducción a C#,
 * Nacho Cabanes
 */
/*-----*/
using System;

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

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

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

```

Que mostraría en pantalla lo siguiente:

```

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

```

Ejercicios propuestos:

- **(4.4.6.3)** Un programa que pida al usuario una frase y muestre sus palabras en orden inverso.
- **(4.4.6.4)** Un programa que pida al usuario varios números separados por espacios y muestre su suma.

4.4.7. Comparación de cadenas

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

```

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

```

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

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

Un programa completo de prueba podría ser así:

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

using System;

public class Ejemplo43c
{
    public static void Main()
    {
        string frase;

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

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

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

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

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

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

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

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

Ejercicios propuestos:

- **(4.4.7.1)** Un programa que pida al usuario cinco frases, las guarde en un array y muestre la "mayor" de ellas (la que aparecería en último lugar en un diccionario).

4.4.8. Una cadena modificable: `StringBuilder`

Si tenemos la necesidad de poder modificar una cadena letra a letra, no podemos usar un "string" convencional, deberemos recurrir a un "StringBuilder", que sí lo permiten pero son algo más complejos de manejar: hay de reservarles espacio con "new" (igual que hacíamos en ciertas ocasiones con los Arrays), y se pueden convertir a una cadena "convencional" usando "ToString":

```
/*
 * Ejemplo en C# nº 44:
 * ejemplo44.cs
 *
 * Cadenas modificables
 * con "StringBuilder"
 *
 * Introduccion a C#,
 * Nacho Cabanes
 */

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

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

```

    cadenaModificable);

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

```

Ejercicios propuestos:

- **(4.4.8.1)** 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".
- **(4.4.8.2)** Un programa que pida tu nombre, tu día de nacimiento y tu mes de nacimiento y lo junte todo en una cadena, separando el nombre de la fecha por una coma y el día del mes por una barra inclinada, así: "Juan, nacido el 31/12".
- **(4.4.8.3)** Crear un juego del ahorcado, en el que un primer usuario introduzca la palabra a adivinar, se muestre esta palabra oculta con guiones (-----) y el programa acepte las letras que introduzca el segundo usuario, cambiando los guiones por letras correctas cada vez que acierte (por ejemplo, a---a-t-). La partida terminará cuando se acierte la palabra por completo o el usuario agote sus 8 intentos.

4.4.9. Recorriendo con "foreach"

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

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

```

/*
 * Ejemplo en C# nº 45:
 */
/* ejemplo45.cs */
/*
 * Ejemplo de "foreach"
 */
/*
 * Introduccion a C#,
 */
/* Nacho Cabanes */
/*
 */

using System;

public class Ejemplo45
{

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

```

```

    }

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

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

}

```

Ejercicios propuestos:

- **(4.4.9.1)** 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".
- **(4.4.9.2)** Un programa que pida al usuario varios números separados por espacios y muestre su suma (como el del ejercicio 4.4.6.4), pero empleando "foreach".

4.5 Ejemplo completo

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

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

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

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

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

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

o bien algo como

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

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

Con todo esto, nuestro fuente quedaría así:

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

public class Ejemplo46
{

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

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

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

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

```

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

```

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

```

public class Ejemplo
{
    ...

    public LanzarJuego () {
        Juego j = new Juego();
        j.ComienzoPartida ();
        ...
    }
}

```

Ejercicio propuesto:

- **(7.1.1)** Crea una nueva versión del ejercicio 5.2.3, en la que métodos y variables no sean "static".

7.2. Arrays de objetos

Es muy frecuente que no nos baste con tener un objeto de cada clase, sino que necesitemos manipular varios objetos pertenecientes a la misma clase. En ese caso, podríamos almacenar todos ellos en un "array". Al declararlo, deberemos reservar memoria primero para el array, y luego para cada uno de los elementos. Por ejemplo, podríamos tener un array de 5 perros, que crearíamos de esta forma:

```

Perro[] misPerros = new Perro[5];
for (byte i = 0; i < 5; i++)
    misPerros[i] = new Perro();

```

Un fuente completo de ejemplo podría ser

```

/*
 * Ejemplo en C# nº 63: *
 * ejemplo63.cs *
 */
/* Quinto ejemplo de clases */
/* Array de objetos */
/* Introducción a C#, */
/* Nacho Cabanes */
/*
using System;

public class Animal
{
    public Animal()
    {
        Console.WriteLine("Ha nacido un animal");
    }
}

public class Perro: Animal

```

```
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}

// ----

public class Ejemplo63
{

    public static void Main()
    {
        Perro[] misPerros = new Perro[5];
        for (byte i = 0; i < 5; i++)
            misPerros[i] = new Perro();
    }
}
```

y su salida en pantalla, parecida a la del ejemplo anterior, sería

```
Ha nacido un animal
Ha nacido un perro
```

Ejercicio propuesto:

- **(7.2.1)** Crea una versión ampliada del ejercicio 6.8.1, en la que no se cree un único objeto de cada clase, sino un array de tres objetos.

Además, existe una peculiaridad curiosa: podemos crear un array de "Animales", pero luego indicar que unos de ellos son perros, otros gatos, etc.,

```
Animal[] misAnimales = new Animal[3];

misAnimales[0] = new Perro();
misAnimales[1] = new Gato();
misAnimales[2] = new GatoSiames();
```

Un ejemplo más detallado:

```
/*
 *----- Ejemplo en C# nº 64:
 *----- ejemplo64.cs
 *
 *----- Ejemplo de clases
 *----- Array de objetos de
 *----- varias subclases
 *
 *----- Introducción a C#,
 *----- Nacho Cabanes
 *----- */

using System;

public class Animal
{
    public Animal()
    {
        Console.WriteLine("Ha nacido un animal");
    }
}

// ----

public class Perro: Animal
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}

// ----

public class Gato: Animal
{
    public Gato()
    {
        Console.WriteLine("Ha nacido un gato");
    }
}

// ----

public class GatoSiames: Gato
{
    public GatoSiames()
    {
        Console.WriteLine("Ha nacido un gato siamés");
    }
}

// -----
```

```

public class Ejemplo64
{
    public static void Main()
    {
        Animal[] misAnimales = new Animal[8];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new GatoSiames();

        for (byte i=3; i<7; i++)
            misAnimales[i] = new Perro();

        misAnimales[7] = new Animal();
    }
}

```

La salida de este programa sería:

```

Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un gato
Ha nacido un animal
Ha nacido un gato
Ha nacido un gato siamés
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal

```

7.3. Funciones virtuales. La palabra "override"

En el ejemplo anterior hemos visto cómo crear un array de objetos, usando sólo la clase base, pero insertando realmente objetos de cada una de las clases derivadas que nos interesaba, y hemos visto que los constructores se llaman correctamente... pero con los métodos puede haber problemas.

Vamos a verlo con un ejemplo, que en vez de tener constructores va a tener un único método "Hablar", que se redefine en cada una de las clases hijas, y después comentaremos qué ocurre al ejecutarlo:

```

/*-----*/
/* Ejemplo en C# nº 65:      */
/* ejemplo65.cs               */

```

11. Punteros y gestión dinámica de memoria

11.1. ¿Por qué usar estructuras dinámicas?

Hasta ahora teníamos una serie de variables que declaramos al principio del programa o de cada función. Estas variables, que reciben el nombre de **ESTÁTICAS**, tienen un tamaño asignado desde el momento en que se crea el programa.

Este tipo de variables son sencillas de usar y rápidas... si sólo vamos a manejar estructuras de datos que no cambien, pero resultan poco eficientes si tenemos estructuras cuyo tamaño no sea siempre el mismo.

Es el caso de una agenda: tenemos una serie de fichas, e iremos añadiendo más. Si reservamos espacio para 10, no podremos llegar a añadir la número 11, estamos limitando el máximo. Una solución sería la de trabajar siempre en el disco: no tenemos límite en cuanto a número de fichas, pero es muchísimo más lento.

Lo ideal sería aprovechar mejor la memoria que tenemos en el ordenador, para guardar en ella todas las fichas o al menos todas aquellas que quepan en memoria.

Una solución "típica" (pero mala) es sobredimensionar: preparar una agenda contando con 1000 fichas, aunque supongamos que no vamos a pasar de 200. Esto tiene varios inconvenientes: se desperdicia memoria, obliga a conocer bien los datos con los que vamos a trabajar, sigue pudiendo verse sobredimensionado, etc.

+

La solución suele ser crear estructuras **DINÁMICAS**, que puedan ir creciendo o disminuyendo según nos interese. En los lenguajes de programación "clásicos", como C y Pascal, este tipo de estructuras se tienen que crear de forma básicamente artesanal, mientras que en lenguajes modernos como C#, Java o las últimas versiones de C++, existen esqueletos ya creados que podemos utilizar con facilidad.

Algunos ejemplos de estructuras de este tipo son:

- Las **pilas**. Como una pila de libros: vamos apilando cosas en la cima, o cogiendo de la cima. Se supone que no se puede tomar elementos de otro sitio que no sea la cima, ni dejarlos en otro sitio distinto. De igual modo, se supone que la pila no tiene un tamaño máximo definido, sino que puede crecer arbitrariamente.
- Las **colas**. Como las del cine (en teoría): la gente llega por un sitio (la cola) y sale por el opuesto (la cabeza). Al igual que antes, supondremos que un elemento no puede entrar a la cola ni salir de ella en posiciones intermedias y que la cola puede crecer hasta un tamaño indefinido.
- Las **listas**, en las que se puede añadir elementos en cualquier posición, y borrarlos de cualquier posición.

Y la cosa se va complicando: en los **árboles** cada elemento puede tener varios sucesores (se parte de un elemento "raíz", y la estructura se va ramificando), etc.

Todas estas estructuras tienen en común que, si se programan correctamente, pueden ir creciendo o decreciendo según haga falta, al contrario que un array, que tiene su tamaño prefijado.

Veremos ejemplos de cómo crear estructuras dinámicas de estos tipos en C#, y después comentaremos los pasos para crear una estructura dinámica de forma "artesanal".

11.2. Una pila en C#

Para crear una pila tenemos preparada la clase Stack. Los métodos habituales que debería permitir una pila son introducir un nuevo elemento en la cima ("apilar", en inglés "push"), y quitar el elemento que hay en la cima ("desapilar", en inglés "pop"). Este tipo de estructuras se suele llamar también con las siglas "LIFO" (Last In First Out: lo último en entrar es lo primero en salir). Para utilizar la clase "Stack" y la mayoría de las que veremos en este tema, necesitamos incluir en nuestro programa una referencia a "System.Collections".

Así, un ejemplo básico que creara una pila, introdujera tres palabras y luego las volviera a mostrar sería:

```
/*
 * Ejemplo en C#
 */
/* pilal.cs */
/*
 */
/* Ejemplo de clase "Stack" */
/*
 */
/* Introduccion a C#, */
/* Nacho Cabanes */
/*
 */

using System;
using System.Collections;

public class ejemploPila1 {
    public static void Main() {
        string palabra;

        Stack miPila = new Stack();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        for (byte i=0; i<3; i++) {
            palabra = (string) miPila.Pop();
            Console.WriteLine( palabra );
        }
    }
}
```

cuyo resultado sería:

```

yo
soy
Hola,

```

La implementación de una pila en C# es algo más avanzada: permite también métodos como:

- "Peek", que mira el valor que hay en la cima, pero sin extraerlo.
- "Clear", que borra todo el contenido de la pila.
- "Contains", que indica si un cierto elemento está en la pila.
- "GetType", para saber de qué tipo son los elementos almacenados en la pila.
- "ToString", que devuelve el elemento actual convertido a un string.
- "ToArray", que devuelve toda la pila convertida a un array.
- "GetEnumerator", que permite usar "enumeradores" para recorrer la pila, una funcionalidad que veremos con algún detalle más adelante.
- También tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

Ejercicios propuestos:

- **(11.2.1)** La "notación polaca inversa" es una forma de expresar operaciones que consiste en indicar los operandos antes del correspondiente operador. Por ejemplo, en vez de "3+4" se escribiría "3 4 +". Es una notación que no necesita paréntesis y que se puede resolver usando una pila: si se recibe un dato numérico, éste se guarda en la pila; si se recibe un operador, se obtienen los dos operandos que hay en la cima de la pila, se realiza la operación y se apila su resultado. El proceso termina cuando sólo hay un dato en la pila. Por ejemplo, "3 4 +" se convierte en: apilar 3, apilar 4, sacar dos datos y sumarlos, guardar 7, terminado. Impleméntalo y comprueba si el resultado de "3 4 6 5 - + * 6 +" es 21.

11.3. Una cola en C#

Podemos crear colas si nos apoyamos en la clase Queue. En una cola podremos introducir elementos por la cabeza ("Enqueue", encolar) y extraerlos por el extremo opuesto, el final de la cola ("Dequeue", desencolar). Este tipo de estructuras se nombran a veces también por las siglas FIFO (First In First Out, lo primero en entrar es lo primero en salir). Un ejemplo básico similar al anterior, que creara una cola, introdujera tres palabras y luego las volviera a mostrar sería:

```

/*
 * Ejemplo en C#
 */
/*
 * Ejemplo de clase "Queue"
 */
/*
 * Introduccion a C#,
 */
/*
 * Nacho Cabanes
 */
/*
 */

```

```

using System;
using System.Collections;

```

```

public class ejemploCola1 {
    public static void Main() {
        string palabra;

        Queue miCola = new Queue();
        miCola.Enqueue("Hola,");
        miCola.Enqueue("soy");
        miCola.Enqueue("yo");

        for (byte i=0; i<3; i++) {
            palabra = (string) miCola.Dequeue();
            Console.WriteLine( palabra );
        }
    }
}

```

que mostraría:

```

Hola,
soy
yo

```

Al igual que ocurría con la pila, la implementación de una cola que incluye C# es más avanzada que eso, con métodos similares a los de antes:

- "Peek", que mira el valor que hay en la cabeza de la cola, pero sin extraerlo.
- "Clear", que borra todo el contenido de la cola.
- "Contains", que indica si un cierto elemento está en la cola.
- "GetType", para saber de qué tipo son los elementos almacenados en la cola.
- "ToString", que devuelve el elemento actual convertido a un string.
- "ToArray", que devuelve toda la pila convertida a un array.
- "GetEnumerator", que permite usar "enumeradores" para recorrer la cola, una funcionalidad que veremos con algún detalle más adelante.
- Al igual que en la pila, también tenemos una propiedad "Count", que nos indica cuántos elementos contiene.

Ejercicios propuestos:

- **(11.3.1)** Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en una cola, luego muestre este contenido en pantalla y finalmente lo vuelque a otro fichero de texto.

11.4. Las listas

Una lista es una estructura dinámica en la que se puede añadir elementos sin tantas restricciones. Es habitual que se puedan introducir nuevos datos en ambos extremos, así como entre dos elementos existentes, o bien incluso de forma ordenada, de modo que cada nuevo dato se introduzca automáticamente en la posición adecuada para que todos ellos queden en orden.

En el caso de C#, no tenemos ninguna clase "List" que represente una lista genérica, pero sí dos variantes especialmente útiles: una lista ordenada ("SortedList") y una lista a cuyos elementos se puede acceder como a los de un array ("ArrayList").

11.4.1. ArrayList

En un ArrayList, podemos añadir datos en la última posición con "Add", insertar en cualquier otra con "Insert", recuperar cualquier elemento usando corchetes, o bien ordenar toda la lista con "Sort". Vamos a ver un ejemplo de la mayoría de sus posibilidades:

```
/*
 *----- Ejemplo en C#
 *----- arrayList1.cs
 *----- Ejemplo de ArrayList
 *----- Introduccion a C#,
 *----- Nacho Cabanes
 *----- */

using System;
using System.Collections;

public class ejemploArrayList1 {

    public static void Main() {

        ArrayList miLista = new ArrayList();
        // Añadimos en orden
        miLista.Add("Hola,");
        miLista.Add("soy");
        miLista.Add("yo");

        // Mostramos lo que contiene
        Console.WriteLine( "Contenido actual:" );
        foreach (string frase in miLista)
            Console.WriteLine( frase );

        // Accedemos a una posición
        Console.WriteLine( "La segunda palabra es: {0}" ,
            miLista[1] );

        // Insertamos en la segunda posición
        miLista.Insert(1, "Como estas?" );

        // Mostramos de otra forma lo que contiene
        Console.WriteLine( "Contenido tras insertar:" );
        for (int i=0; i<miLista.Count; i++)
    }
}
```

```

Console.WriteLine( miLista[i] );

// Buscamos un elemento
Console.WriteLine( "La palabra \"yo\" está en la posición {0}",
    miLista.IndexOf("yo") );

// Ordenamos
miLista.Sort();

// Mostramos lo que contiene
Console.WriteLine( "Contenido tras ordenar");
foreach (string frase in miLista)
    Console.WriteLine( frase );

// Buscamos con búsqueda binaria
Console.WriteLine( "Ahora \"yo\" está en la posición {0}",
    miLista.BinarySearch("yo") );

// Invertimos la lista
miLista.Reverse();

// Borramos el segundo dato y la palabra "yo"
miLista.RemoveAt(1);
miLista.Remove("yo");

// Mostramos nuevamente lo que contiene
Console.WriteLine( "Contenido dar la vuelta y tras eliminar dos:" );
foreach (string frase in miLista)
    Console.WriteLine( frase );

// Ordenamos y vemos dónde iría un nuevo dato
miLista.Sort();
Console.WriteLine( "La frase \"Hasta Luego\"..." );
int posicion = miLista.BinarySearch("Hasta Luego");
if (posicion >= 0)
    Console.WriteLine( "Está en la posición {0}", posicion );
else
    Console.WriteLine( "No está. El dato inmediatamente mayor es el {0}: {1}",
        ~posicion, miLista[~posicion] );

}
}

```

El resultado de este programa es:

```

Contenido actual:
Hola,
soy
yo
La segunda palabra es: soy
Contenido tras insertar:
Hola,
Como estas?
soy
yo

```

```

La palabra "yo" está en la posición 3
Contenido tras ordenar
Como estas?
Hola,
soy
yo
Ahora "yo" está en la posición 3
Contenido dar la vuelta y tras eliminar dos:
Hola,
Como estas?
La frase "Hasta Luego"...
No está. El dato inmediatamente mayor es el 1: Hola,

```

Casi todo debería resultar fácil de entender, salvo quizá el símbolo `~`. Esto se debe a que `BinarySearch` devuelve un número negativo cuando el texto que buscamos no aparece, pero ese número negativo tiene un significado: es el "valor complementario" de la posición del dato inmediatamente mayor (es decir, el dato cambiando los bits 0 por 1 y viceversa). En el ejemplo anterior, "posición" vale -2, lo que quiere decir que el dato no existe, y que el dato inmediatamente mayor está en la posición 1 (que es el "complemento a 2" del número -2, que es lo que indica la expresión "`~posición`"). En el apéndice 3 de este texto hablaremos de cómo se representan internamente los números enteros, tanto positivos como negativos, y entonces se verá con detalle en qué consiste el "complemento a 2".

A efectos prácticos, lo que nos interesa es que si quisiéramos insertar la frase "Hasta Luego", su posición correcta para que todo el `ArrayList` permaneciera ordenado sería la 1, que viene indicada por "`~posicion`".

Veremos los operadores a nivel de bits, como `~`, en el tema 13, que estará dedicado a otras características avanzadas de C#.

Ejercicios propuestos:

- **(11.4.1.1)** Crea un programa que lea el contenido de un fichero de texto, lo almacene línea por línea en un `ArrayList`, luego muestre en pantalla las líneas impares (primera, tercera, etc.) y finalmente vuelque a otro fichero de texto las líneas pares (segunda, cuarta, etc.).
- **(11.4.1.2)** Crea una nueva versión de la "bases de datos de ficheros" (ejemplo 46), pero usando `ArrayList` en vez de un array convencional.

11.4.2. `SortedList`

En un `SortedList`, los elementos están formados por una pareja: una clave y un valor (como en un diccionario: la palabra y su definición). Se puede añadir elementos con `"Add"`, o acceder a los elementos mediante su índice numérico (con `"GetKey"`) o mediante su clave (sabiendo en qué posición se encuentra una clave con `"IndexOfKey"`), como en este ejemplo:

```

/*
/* Ejemplo en C#
/* sortedList1.cs */

```

```

/*
/* Ejemplo de SortedList: */
/* Diccionario esp-ing */
/*
/* Introducción a C#,
/* Nacho Cabanes */
/*
using System;
using System.Collections;
public class ejemploSortedList {

    public static void Main() {

        // Creamos e insertamos datos
        SortedList miDiccio = new SortedList();
        miDiccio.Add("holá", "hello");
        miDiccio.Add("adiós", "good bye");
        miDiccio.Add("hasta luego", "see you later");

        // Mostramos los datos
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}" ,
            miDiccio.Count );
        Console.WriteLine( "Lista de palabras y su significado:" );
        for (int i=0; i<miDiccio.Count; i++) {
            Console.WriteLine( "{0} = {1}" ,
                miDiccio.GetKey(i), miDiccio.GetByIndex(i) );
        }
        Console.WriteLine( "Traducción de \"holá\": {0}" ,
            miDiccio.GetByIndex( miDiccio.IndexOfKey("holá") ) );
        Console.WriteLine( "Que también se puede obtener con corchetes: {0}" ,
            miDiccio["holá"]);
    }
}

```

Su resultado sería

```

Cantidad de palabras en el diccionario: 3
Lista de palabras y su significado:
adiós = good bye
hasta luego = see you later
holá = hello
Traducción de "holá": hello

```

Otras posibilidades de la clase `SortedList` son:

- "Contains", para ver si la lista contiene una cierta clave.
- "ContainsValue", para ver si la lista contiene un cierto valor.
- "Remove", para eliminar un elemento a partir de su clave.
- "RemoveAt", para eliminar un elemento a partir de su posición.
- "SetByIndex", para cambiar el valor que hay en una cierta posición.

Ejercicios propuestos:

- **(11.4.2.1)** Crea un traductor básico de C# a Pascal, que tenga las traducciones almacenadas en una SortedList (por ejemplo, "{" se convertirá a "begin", "}" se convertirá a "begin", "WriteLine" se convertirá a "WriteLn", "ReadLine" se convertirá a "ReadLn", "void" se convertirá a "procedure" y "Console." se convertirá a una cadena vacía.

11.5. Las "tablas hash"

En una "tabla hash", los elementos están formados por una pareja: una clave y un valor, como en un SortedList, pero la diferencia está en la forma en que se manejan internamente estos datos: la "tabla hash" usa una "función de dispersión" para colocar los elementos, de forma que no se pueden recorrer secuencialmente, pero a cambio el acceso a partir de la clave es **muy rápido**, más que si hacemos una búsqueda secuencial (como en un array) o binaria (como en un ArrayList ordenado).

Un ejemplo de diccionario, parecido al anterior (que es más rápido de consultar para un dato concreto, pero que no se puede recorrer en orden), podría ser:

```
/*
/* Ejemplo en C#
/* HashTable1.cs
/*
/*
/* Ejemplo de HashTable:
/* Diccionario de inform.
/*
/*
/* Introduccion a C#,
/* Nacho Cabanes
/*
*/
/*-----*/



using System;
using System.Collections;
public class ejemploHashTable {

    public static void Main() {

        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos algún dato
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}",
            miDiccio.Count );
        try {
            Console.WriteLine( "El significado de PC es: {0}",
                miDiccio["pc"]);
        } catch (Exception e) {
            Console.WriteLine( "No existe esa palabra!" );
        }
    }
}
```

que escribiría en pantalla:

```
Cantidad de palabras en el diccionario: 3
El significado de PC es: personal computer
```

Si un elemento que se busca no existe, se lanzaría una excepción, por lo que deberíamos controlarlo con un bloque try..catch. Lo mismo ocurre si intentamos introducir un dato que ya existe. Una alternativa a usar try..catch es comprobar si el dato ya existe, con el método "Contains" (o su sinónimo "ContainsKey"), como en este ejemplo:

```
/*
 * Ejemplo en C#
 * HashTable2.cs
 *
 * Ejemplo de HashTable 2:
 * Diccionario de inform.
 *
 * Introduccion a C#,
 * Nacho Cabanes
 */

using System;
using System.Collections;
public class ejemploHashTable2 {

    public static void Main() {

        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos algún dato
        Console.WriteLine( "Cantidad de palabras en el diccionario: {0}" ,
            miDiccio.Count );
        if (miDiccio.ContainsKey("pc"))
            Console.WriteLine( "El significado de PC es: {0}" ,
                miDiccio["pc"] );
        else
            Console.WriteLine( "No existe la palabra PC" );
    }
}
```

Otras posibilidades son: borrar un elemento ("Remove"), vaciar toda la tabla ("Clear"), o ver si contiene un cierto valor ("ContainsValue", mucho más lento que buscar entre las claves con "Contains").

Una tabla hash tiene una cierta capacidad inicial, que se amplía automáticamente cuando es necesario. Como la tabla hash es mucho más rápida cuando está bastante vacía que cuando está casi llena, podemos usar un constructor alternativo, en el que se le indica la capacidad inicial que queremos, si tenemos una idea aproximada de cuántos datos vamos a guardar:

```
Hashtable miDiccio = new Hashtable(500);
```

Ejercicios propuestos:

- **(11.5.1)** Crea una versión alternativa del ejercicio 11.4.2.1, pero que tenga las traducciones almacenadas en una tabla Hash.

11.6. Los "enumeradores"

Un enumerador es una estructura auxiliar que permite recorrer las estructuras dinámicas de forma secuencial. Casi todas ellas contienen un método GetEnumerator, que permite obtener un enumerador para recorrer todos sus elementos. Por ejemplo, en una tabla hash podríamos hacer:

```
/*
 * Ejemplo en C#
 */
/* HashTable3.cs */
/*
 */
/* Ejemplo de HashTable */
/*
 * y enumerador
 */
/*
 */
/* Introduccion a C#,
 * Nacho Cabanes
 */
/*
 */

using System;
using System.Collections;
public class ejemploHashTable3 {

    public static void Main() {

        // Creamos e insertamos datos
        Hashtable miDiccio = new Hashtable();
        miDiccio.Add("byte", "8 bits");
        miDiccio.Add("pc", "personal computer");
        miDiccio.Add("kilobyte", "1024 bytes");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IDictionaryEnumerator miEnumerador = miDiccio.GetEnumerator();
        while (miEnumerador.MoveNext())
            Console.WriteLine("{0} = {1}",
                miEnumerador.Key, miEnumerador.Value);
    }
}
```

cuyo resultado es

```
Contenido:
pc = personal computer
byte = 8 bits
kilobyte = 1024 bytes
```

Como se puede ver, los enumeradores tendrán un método "MoveNext", que intenta moverse al siguiente elemento y devuelve "false" si no lo consigue. En el caso de las tablas hash, que tiene dos campos (clave y valor), el enumerador a usar será un "enumerador de diccionario" (IDictionaryEnumerator), que contiene los campos Key y Value.

Como se ve en el ejemplo, es habitual que no obtengamos la lista de elementos en el mismo orden en el que los introdujimos, debido a que se colocan siguiendo la función de dispersión.

Para las colecciones "normales", como las pilas y las colas, el tipo de Enumerador a usar será un IEnumerator, con un campo Current para saber el valor actual:

```
/*
 * Ejemplo en C#
 * pila2.cs
 *
 * Ejemplo de clase "Stack"
 * y enumerador
 *
 * Introduccion a C#,
 * Nacho Cabanes
 */

using System;
using System.Collections;

public class ejemploPila1 {
    public static void Main() {
        Stack miPila = new Stack();
        miPila.Push("Hola,");
        miPila.Push("soy");
        miPila.Push("yo");

        // Mostramos todos los datos
        Console.WriteLine("Contenido:");
        IEnumerator miEnumerador = miPila.GetEnumerator();
        while (miEnumerador.MoveNext())
            Console.WriteLine("{0}", miEnumerador.Current);
    }
}
```

que escribiría

```
Contenido:
yo
soy
Hola,
```

Nota: los "enumeradores" existen también en otras plataformas, como Java, aunque allí reciben el nombre de "iteradores".

Se puede saber más sobre las estructuras dinámicas que hay disponibles en la plataforma .Net consultando la referencia en línea de MSDN (mucha de la cual está sin traducir al español):

[http://msdn.microsoft.com/es-es/library/system.collections\(en-us,VS.71\).aspx#](http://msdn.microsoft.com/es-es/library/system.collections(en-us,VS.71).aspx#)

11.7. Cómo "imitar" una pila usando "arrays"

Las estructuras dinámicas se pueden imitar usando estructuras estáticas sobredimensionadas, y esto puede ser un ejercicio de programación interesante. Por ejemplo, podríamos imitar una pila dando los siguientes pasos:

- Utilizamos internamente un array más grande que la cantidad de datos que esperemos que vaya a almacenar la pila.
- Creamos una función "Apilar", que añade en la primera posición libre del array (initialmente la 0) y después incrementa esa posición, para que el siguiente dato se introduzca a continuación.
- Creamos también una función "Desapilar", que devuelve el dato que hay en la última posición, y que disminuye el contador que indica la posición, de modo que el siguiente dato que se obtuviera sería el que se introdujo con anterioridad a éste.

El fuente podría ser así:

```
/*
 * Ejemplo en C#
 */
/* pilaEstatica.cs */
/*
 * Ejemplo de clase "Pila"
 */
/* basada en un array */
/*
 */
/* Introducción a C#,
 */
/* Nacho Cabanes
*/
/*-----*/
```

```
using System;
using System.Collections;

public class PilaString {
    string[] datosPila;
    int posicionPila;
    const int MAXPILA = 200;

    public static void Main() {
        string palabra;

        PilaString miPila = new PilaString();
        miPila.Apilar("Hola,");
        miPila.Apilar("soy");
        miPila.Apilar("yo");

        for (byte i=0; i<3; i++) {
            palabra = (string) miPila.Desapilar();
        }
    }
}
```

```

        Console.WriteLine( palabra );
    }

}

// Constructor
public PilaString() {
    posicionPila = 0;
    datosPila = new string[MAXPILA];
}

// Añadir a la pila: Apilar
public void Apilar(string nuevoDatos) {
    if (posicionPila == MAXPILA)
        Console.WriteLine("Pila llena!");
    else {
        datosPila[posicionPila] = nuevoDatos;
        posicionPila++;
    }
}

// Extraer de la pila: Desapilar
public string Desapilar() {
    if (posicionPila < 0)
        Console.WriteLine("Pila vacia!");
    else {
        posicionPila--;
        return datosPila[posicionPila];
    }
    return null;
}

} // Fin de la clase

```

Ejercicios propuestos:

- **(11.7.1)** Usando esta misma estructura de programa, crear una clase "Cola", que permita introducir datos (números enteros) y obtenerlos en modo FIFO (el primer dato que se introduzca debe ser el primero que se obtenga). Debe tener un método "Encolar" y otro "Desencolar".
- **(11.7.2)** Crear una clase "ListaOrdenada", que almacene un único dato (no un par clave-valor como los SortedList). Debe contener un método "Insertar", que añadirá un nuevo dato en orden en el array, y un "Extraer(n)", que obtenga un elemento de la lista (el número "n"). Deberá almacenar "strings".
- **(11.7.3)** Crea una pila de "doubles", usando internamente un ArrayList en vez de un array.
- **(11.7.4)** Crea una cola que almacene un bloque de datos (struct, con los campos que tú elijas) usando un ArrayList.
- **(11.7.5)** Crea una lista ordenada (de "strings") usando un ArrayList.

11.8. Introducción a los "generics"

Una ventaja, pero también a la vez un inconveniente, de las estructuras dinámicas que hemos visto, es que permiten guardar datos de cualquier tipo, incluso datos de distinto tipo en una

13.3. Enumeraciones

Cuando tenemos varias constantes, cuyos valores son números enteros, hasta ahora estamos dando los valores uno por uno, así:

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

Hay una forma alternativa de hacerlo, especialmente útil si son números enteros consecutivos. Se trata de **enumerarlos**:

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

(Al igual que las constantes de cualquier otro tipo, se puede escribir en mayúsculas para recordar "de un vistazo" que son constantes, no variables)

La primera constante valdrá 0, y las demás irán aumentando de una en una, de modo que en nuestro caso valen:

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

Si queremos que los valores no sean exactamente estos, podemos dar valor a cualquiera de las constantes, y las siguientes irán aumentando de uno en uno. Por ejemplo, si escribimos

```
enum diasSemana { LUNES=1, MARTES, MIERCOLES, JUEVES=6, VIERNES,
                  SABADO=10, DOMINGO };
```

Ahora sus valores son:

```
LUNES = 1, MARTES = 2, MIERCOLES = 3, JUEVES = 6, VIERNES = 7,
SABADO = 10, DOMINGO = 11
```

Un ejemplo básico podría ser

```
/*
 * Ejemplo en C#
 */
/* enum.cs */
/*
 */
/* Ejemplo de enumeraciones */
/*
 */
/* Introducción a C#,
 */
/* Nacho Cabanes */
/*
*/
using System;

public class enumeraciones
{
    enum diasSemana { LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO,
```

```

DOMINGO };

public static void Main()
{
    Console.WriteLine("En la enumeracion, el miércoles tiene el valor: {0} ",
        diasSemana.MIERCOLES);
    Console.WriteLine("que equivale a: {0}",
        (int) diasSemana.MIERCOLES);

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

    Console.WriteLine("En las constantes, el miércoles tiene el valor: {0}",
        MIERCOLES);
}

```

y su resultado será:

```

En la enumeracion, el miércoles tiene el valor: MIERCOLES que equivale a: 2
En las constantes, el miércoles tiene el valor: 2

```

Nosotros hemos usado enumeraciones muchas veces hasta ahora, sin saber realmente que lo estábamos haciendo. Por ejemplo, el modo de apertura de un fichero (FileMode) es una enumeración, por lo que escribimos FileMode.Open. También son enumeraciones los códigos de color de la consola (como ConsoleColor.Red) y las teclas de la consola (como ConsoleKey.Escape).

Nota: las enumeraciones existen también en otros lenguajes como C y C++, pero la sintaxis es ligeramente distinta: en C# es necesario indicar el nombre de la enumeración cada vez que se usen sus valores (como en díasSemana.MIERCOLES), mientras que en C se usa sólo el valor (MIERCOLES).

Ejercicios propuestos

- **(13.3.1)** Crea una versión de la agenda con colores (ejercicio 12.2.2) en la que las opciones sean parte de una enumeración.

13.4. Propiedades

Hasta ahora estábamos siguiendo la política de que los atributos de una clase sean privados, y se acceda a ellos a través de métodos "get" (para leer su valor) y "set" (para cambiarlo). En el caso de C#, existe una forma alternativa de conseguir el mismo efecto, empleando las llamadas "propiedades", que tienen una forma abreviada de escribir sus métodos "get" y "set":

```

/*
 *-----*
 * Ejemplo en C#
 *-----*
 * propiedades.cs
 *-----*
 * Ejemplo de propiedades
 */

```

```
/* Introducción a C#,          */
/*      Nacho Cabanes          */
/*-----*/  
  
using System;  
  
public class EjemploPropiedades  
{  
  
    // -----  
  
    // Un atributo convencional, privado  
    private int altura = 0;  
  
    // Para ocultar detalles, leemos su valor con un "get"  
    public int GetAltura()  
    {  
        return altura;  
    }  
  
    // Y lo fijamos con un "set"  
    public void SetAltura(int nuevoValor)  
    {  
        altura = nuevoValor;  
    }  
  
    // -----  
  
    // Otro atributo convencional, privado  
    private int anchura = 0;  
  
    // Oculta mediante una "propiedad"  
    public int Anchura  
    {  
        get  
        {  
            return anchura;  
        }  
  
        set  
        {  
            anchura = value;  
        }  
    }  
  
    // -----  
  
    // El "Main" de prueba  
    public static void Main()  
    {  
  
        EjemploPropiedades ejemplo  
        = new EjemploPropiedades();  
  
        ejemplo.SetAltura(5);  
        Console.WriteLine("La altura es {0}",  
                         ejemplo.GetAltura());  
    }  
}
```

```

        ejemplo.Anchura = 6;
        Console.WriteLine("La anchura es {0}",
            ejemplo.Anchura);
    }
}

```

Al igual que ocurría con las enumeraciones, ya hemos usado "propiedades" anteriormente, sin saberlo: la longitud ("Length") de una cadena, el tamaño ("Length") y la posición actual ("Position") en un fichero, el título ("Title") de una ventana en consola, etc.

Una curiosidad: si una propiedad tiene un "get", pero no un "set", será una propiedad de sólo lectura, no podremos hacer cosas como "Anchura = 4", porque el programa no compilaría. De igual modo, se podría crear una propiedad de sólo escritura, definiendo su "set" pero no su "get".

Ejercicios propuestos

- **(13.4.1)** Añade al ejercicio de los trabajadores (6.7.1) una propiedad "nombre", con sus correspondientes "get" y "set".

13.5. Parámetros de salida (*out*)

Hemos hablado de dos tipos de parámetros de una función: parámetros por valor (que no se pueden modificar) y parámetros por referencia ("ref", que sí se pueden modificar). Un uso habitual de los parámetros por referencia es devolver más de un valor a la salida de una función. Para ese uso, en C# existe otra alternativa: los parámetros de salida. Se indican con la palabra "out" en vez de "ref", y no exigen que las variables tengan un valor inicial:

```

public void ResolverEcuacionSegundoGrado(
    float a, float b, float c,
    out float x1, out float x2)

```

Ejercicios propuestos

- **(13.5.1)** Crea una nueva versión del ejercicio que resuelve ecuaciones de segundo grado (5.9.2.2), usando parámetros "out".

13.6. Introducción a las expresiones regulares.

Las "expresiones regulares" permiten hacer comparaciones mucho más abstractas que si se usa un simple "IndexOf". Por ejemplo, podemos comprobar con una orden breve si todos los caracteres de una cadena son numéricos, o si empieza por mayúscula y el resto son minúsculas, etc.

Vamos a ver solamente un ejemplo con un caso habitual: comprobar si una cadena es numérica, alfabética o alfanumérica. Las ideas básicas son:

```

        Console.WriteLine("\nVamos a abrir...");  

        p.Abrir();  

        p.MostrarEstado();  

    }  

}

```

Y lo compilaríamos con:

```
gmcs ejemplo59b.cs ejemplo59c.cs -out:ejemplo59byc.exe
```

Aun así, para estos proyectos formados por varias clases, lo ideal es usar algún entorno más avanzado, como SharpDevelop o VisualStudio, que permitan crear todas las clases con comodidad, saltar de una clase a clase otra rápidamente, que marquen dentro del propio editor la línea en la que están los errores... por eso, al final de este tema tendrás un apartado con una introducción al uso de SharpDevelop.

Ejercicio propuesto:

- **(6.2.2)** Modificar el fuente del ejercicio anterior, para dividirlo en dos ficheros: Crear una clase llamada Persona, en el fichero "persona.cs". Esta clase deberá tener un atributo "nombre", de tipo string. También deberá tener un método "SetNombre", de tipo void y con un parámetro string, que permita cambiar el valor del nombre. Finalmente, también tendrá un método "Saludar", que escribirá en pantalla "Hola, soy " seguido de su nombre. Crear también una clase llamada PruebaPersona, en el fichero "pruebaPersona.cs". Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona, les asignará un nombre y les pedirá que saluden.

6.3. La herencia. Visibilidad

Vamos a ver ahora cómo definir una nueva clase de objetos a partir de otra ya existente. Por ejemplo, vamos a crear una clase "Porton" a partir de la clase "Puerta". Un portón tendrá las mismas características que una puerta (ancho, alto, color, abierto o no), pero además se podrá bloquear, lo que supondrá un nuevo atributo y nuevos métodos para bloquear y desbloquear:

```

public class Porton: Puerta  

{  

    bool bloqueada;  

    public void Bloquear()  

    {  

        bloqueada = true;  

    }  

    public void Desbloquear()  

    {  

        bloqueada = false;
    }
}

```

```

    }
}
```

Con "public class Porton: Puerta" indicamos que Porton debe "heredar" todo lo que ya habíamos definido para Puerta. Por eso, no hace falta indicar nuevamente que un Portón tendrá un cierto ancho, o un color, o que se puede abrir: todo eso lo tiene por ser un "descendiente" de Puerta.

No tenemos por qué heredar todo; también podemos "redefinir" algo que ya existía. Por ejemplo, nos puede interesar que "MostrarEstado" ahora nos diga también si la puerta está bloqueada. Para eso, basta con volverlo a declarar y añadir la palabra "**new**" para indicar al compilador de C# que sabemos que ya existe ese método y que sabemos seguro que lo queremos redefinir:

```

public new void MostrarEstado()
{
    Console.WriteLine("Ancho: {0}", ancho);
    Console.WriteLine("Alto: {0}", alto);
    Console.WriteLine("Color: {0}", color);
    Console.WriteLine("Abierta: {0}", abierta);
    Console.WriteLine("Bloqueada: {0}", bloqueada);
}
```

Aun así, esto todavía no funciona: los atributos de una Puerta, como el "ancho" y el "alto" estaban declarados como "privados" (es lo que se considera si no decimos lo contrario), por lo que no son accesibles desde ninguna otra clase, ni siquiera desde Porton.

La solución más razonable no es declararlos como "public", porque no queremos que sean accesibles desde cualquier sitio. Sólo querríamos que esos datos estuvieran disponibles para todos los tipos de Puerta, incluyendo sus "descendientes", como un Porton. Esto se puede conseguir usando otro método de acceso: "**protected**". Todo lo que declaremos como "protected" será accesible por las clases derivadas de la actual, pero por nadie más:

```

public class Puerta
{
    protected int ancho;      // Ancho en centimetros
    protected int alto;       // Alto en centimetros
    protected int color;      // Color en formato RGB
    protected bool abierta;   // Abierta o cerrada

    public void Abrir()
    ...
}
```

(Si quisieramos dejar claro que algún elemento de una clase debe ser totalmente privado, podemos usar la palabra "**private**", en vez de "public" o "protected").

Un fuente completo que declarase la clase Puerta, la clase Porton a partir de ella, y que además contuviese un pequeño "Main" de prueba podría ser:

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

```
/* Ejemplo en C# nº 60:      */
/* ejemplo60.cs               */
/*                               */
/* Segundo ejemplo de         */
/* clases: herencia          */
/*                               */
/* Introduccion a C#,          */
/* Nacho Cabanes              */
/*-----*/
using System;

// -----
public class Puerta
{

    protected int ancho;      // Ancho en centimetros
    protected int alto;        // Alto en centimetros
    protected int color;       // Color en formato RGB
    protected bool abierta;    // Abierta o cerrada

    public void Abrir()
    {
        abierta = true;
    }

    public void Cerrar()
    {
        abierta = false;
    }

    public void MostrarEstado()
    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
    }
} // Final de la clase Puerta

// -----
public class Porton: Puerta
{

    bool bloqueada;

    public void Bloquear()
    {
        bloqueada = true;
    }

    public void Desbloquear()
    {
        bloqueada = false;
    }

    public new void MostrarEstado()
    {
        Console.WriteLine("Puerta");
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
        Console.WriteLine("Bloqueada: {0}", bloqueada);
    }
}
```

```

    {
        Console.WriteLine("Ancho: {0}", ancho);
        Console.WriteLine("Alto: {0}", alto);
        Console.WriteLine("Color: {0}", color);
        Console.WriteLine("Abierta: {0}", abierta);
        Console.WriteLine("Bloqueada: {0}", bloqueada);
    }

} // Final de la clase Porton

// -----
public class Ejemplo60
{

    public static void Main()
    {
        Porton p = new Porton();

        Console.WriteLine("Valores iniciales...");
        p.MostrarEstado();

        Console.WriteLine("\nVamos a bloquear...");
        p.Bloquear();
        p.MostrarEstado();

        Console.WriteLine("\nVamos a desbloquear y a abrir...");
        p.Desbloquear();
        p.Abrir();
        p.MostrarEstado();
    }
}

```

Ejercicios propuestos:

- **(6.3.1)** Ampliar las clases del ejercicio 6.2.2, creando un nuevo proyecto con las siguientes características: La clase Persona no cambia. Se creará una nueva clase PersonaIngresa, en el fichero "personaIngresa.cs". Esta clase deberá heredar las características de la clase "Persona", y añadir un método "TomarTe", de tipo void, que escribirá en pantalla "Estoy tomando té". Crear también una clase llamada Prueba-Persona2, en el fichero "pruebaPersona2.cs". Esta clase deberá contener sólo la función Main, que creará dos objetos de tipo Persona y uno de tipo PersonaIngresa, les asignará un nombre, les pedirá que saluden y pedirá a la persona ingresa que tome té.
- **(6.3.2)** Ampliar las clases del ejercicio 6.3.1, creando un nuevo proyecto con las siguientes características: La clase Persona no cambia. La clase PersonaIngresa se modificará para que redefina el método "Saludar", para que escriba en pantalla "Hi, I am " seguido de su nombre. Se creará una nueva clase PersonaItaliana, en el fichero "personaItaliana.cs". Esta clase deberá heredar las características de la clase "Persona", pero redefinir el método "Saludar", para que escriba en pantalla "Ciao". Crear también una clase llamada PruebaPersona3, en el fichero " pruebaPersona3.cs". Esta clase deberá contener sólo la función Main, que creará un objeto de tipo Persona, dos de tipo PersonaIngresa, uno de tipo PersonaItaliana, les asignará un nombre, les pedirá que saluden y pedirá a la persona ingresa que tome té.

Ejercicios propuestos:

- **(6.5.1)** Ampliar las clases del ejercicio 6.4.1, para que todas ellas contengan constructores. Los constructores de casi todas las clases estarán vacíos, excepto del de "Calefactor", que prefijará una temperatura de 25 grados.

6.6. Polimorfismo y sobrecarga

Esos dos constructores "Puerta()" y "Puerta(int ancho, int alto)", que se llaman igual pero reciben distintos parámetros, y se comportan de forma que puede ser distinta, son ejemplos de "**polimorfismo**" (funciones que tienen el mismo nombre, pero distintos parámetros, y que quizás no se comporten de igual forma).

Un concepto muy relacionado con el polimorfismo es el de "**sobrecarga**": dos funciones están sobrecargadas cuando se llaman igual, reciben el mismo número de parámetros, pero se aplican a objetos distintos, así:

```
puerta.Abrir ();
libro.Abrir ();
```

En este caso, la función "Abrir" está sobrecargada: se usa tanto para referirnos a abrir un libro como para abrir una puerta. Se trata de dos acciones que no son exactamente iguales, que se aplican a objetos distintos, pero que se llaman igual.

Ejercicios propuestos:

- **(6.6.1)** Añade a la clase "Ventana" un nuevo método Abrir, que reciba un parámetro, que será un número del 0 al 100 que indique hasta qué punto se debe abrir la ventana (100=100%, abierta; 0=0%, cerrada). Crea los atributos auxiliares que necesites para reflejar esa información.

6.7. Orden de llamada de los constructores

Cuando creamos objetos de una clase derivada, antes de llamar a su constructor se llama a los constructores de las clases base, empezando por la más general y terminando por la más específica. Por ejemplo, si creamos una clase "GatoSiamés", que deriva de una clase "Gato", que a su vez procede de una clase "Animal", el orden de ejecución de los constructores sería: Animal, Gato, GatoSiames, como se ve en este ejemplo:

```
/*
 * Ejemplo en C# nº 62:
 * ejemplo62.cs
 *
 * Cuarto ejemplo de clases
 * Constructores y herencia
 *
 * Introducción a C#
 * Nacho Cabanes
 */
```

```
using System;

public class Animal
{
    public Animal()
    {
        Console.WriteLine("Ha nacido un animal");
    }
}

// ----

public class Perro: Animal
{
    public Perro()
    {
        Console.WriteLine("Ha nacido un perro");
    }
}

// ----

public class Gato: Animal
{
    public Gato()
    {
        Console.WriteLine("Ha nacido un gato");
    }
}

// ----

public class GatoSiames: Gato
{
    public GatoSiames()
    {
        Console.WriteLine("Ha nacido un gato siamés");
    }
}

// ----

public class Ejemplo62
{
    public static void Main()
    {
        Animal a1      = new Animal();
        GatoSiames a2 = new GatoSiames();
        Perro a3      = new Perro();
        Gato a4       = new Gato();
    }
}
```

- **(7.3.2)** Crea una variante del ejercicio 7.3.1, en la que se cree un único array "de trabajadores", que contenga un objeto de cada clase.
- **(7.3.3)** Amplía el ejercicio 7.3.2, para añadir un método "Hablar" en cada clase, redefiniéndolo en las clases hijas usando "new" y probándolo desde "Main".
- **(7.3.4)** Crea una variante del ejercicio 7.3.3, que use "override" en vez de "new".

7.4. Llamando a un método de la clase "padre"

Puede ocurrir que en un método de una clase hija no nos interese redefinir por completo las posibilidades del método equivalente, sino ampliarlas. En ese caso, no hace falta que volvamos a teclear todo lo que hacía el método de la clase base, sino que podemos llamarlo directamente, precediéndolo de la palabra "base". Por ejemplo, podemos hacer que un Gato Siamés hable igual que un Gato normal, pero diciendo "Pfff" después, así:

```
public new void Hablar()
{
    base.Hablar();
    Console.WriteLine("Pfff");
}
```

Este podría ser un fuente completo:

```
/*
 * Ejemplo en C# nº 67:
 * ejemplo67.cs
 *
 * Ejemplo de clases
 * Llamar a la superclase
 *
 * Introducción a C#,
 * Nacho Cabanes
 */

using System;

public class Animal
{

}

// ----

public class Gato : Animal
{
    public void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}

// -----
```

```

public class GatoSiames: Gato
{
    public new void Hablar()
    {
        base.Hablar();
        Console.WriteLine("Pfff");
    }
}

// ----

public class Ejemplo67
{
    public static void Main()
    {
        Gato miGato = new Gato();
        GatoSiames miGato2 = new GatoSiames();

        miGato.Hablar();
        Console.WriteLine(); // Linea en blanco
        miGato2.Hablar();
    }
}

```

Su resultado sería

Miauuu

Miauuu

Pfff

También podemos llamar a un **constructor** de la clase base desde un constructor de una clase derivada. Por ejemplo, si tenemos una clase "RectanguloRelleno" que hereda de otra clase "Rectangulo" y queremos que el constructor de "RectanguloRelleno" que recibe las coordenadas "x" e "y" se base en el constructor equivalente de la clase "Rectangulo", lo haríamos así:

```

public RectanguloRelleno (int x, int y )
    : base (x, y)
{
    // Pasos adicionales
    // que no da un rectangulo "normal"
}

```

(Si no hacemos esto, el constructor de RectanguloRelleno se basaría en el constructor sin parámetros de Rectangulo, no en el que tiene x e y como parámetros).

Ejercicios propuestos:

- **(7.4.1)** Crea una versión ampliada del ejercicio 7.3.4, en la que el método "Hablar" de todas las clases hijas se apoye en el de la clase "Trabajador".

```

public class Ejemplo64
{
    public static void Main()
    {
        Animal[] misAnimales = new Animal[8];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new GatoSiames();

        for (byte i=3; i<7; i++)
            misAnimales[i] = new Perro();

        misAnimales[7] = new Animal();
    }
}

```

La salida de este programa sería:

```

Ha nacido un animal
Ha nacido un perro
Ha nacido un animal
Ha nacido un gato
Ha nacido un animal
Ha nacido un gato
Ha nacido un gato siamés
Ha nacido un animal
Ha nacido un perro
Ha nacido un animal

```

7.3. Funciones virtuales. La palabra "override"

En el ejemplo anterior hemos visto cómo crear un array de objetos, usando sólo la clase base, pero insertando realmente objetos de cada una de las clases derivadas que nos interesaba, y hemos visto que los constructores se llaman correctamente... pero con los métodos puede haber problemas.

Vamos a verlo con un ejemplo, que en vez de tener constructores va a tener un único método "Hablar", que se redefine en cada una de las clases hijas, y después comentaremos qué ocurre al ejecutarlo:

```

/*-----*/
/* Ejemplo en C# nº 65:      */
/* ejemplo65.cs               */

```

```

/*
 * Ejemplo de clases
 * Array de objetos de
 *   varias subclases con
 *   metodos
 *
 * Introduccion a C#,
 * Nacho Cabanes
 */
// ----

using System;

public class Animal
{
    public void Hablar()
    {
        Console.WriteLine("Estoy comunicándome...");
    }
}

// ----

public class Perro: Animal
{
    public new void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}

// ----

public class Gato: Animal
{
    public new void Hablar()
    {
        Console.WriteLine("Miauuu");
    }
}

// ----

public class Ejemplo65
{
    public static void Main()
    {
        // Primero creamos un animal de cada tipo
        Perro miPerro = new Perro();
        Gato miGato = new Gato();
        Animal miAnimal = new Animal();

        miPerro.Hablar();
        miGato.Hablar();
        miAnimal.Hablar();
    }
}

```

```
// Línea en blanco, por legibilidad
Console.WriteLine();

// Ahora los creamos desde un array
Animal[] misAnimales = new Animal[3];

misAnimales[0] = new Perro();
misAnimales[1] = new Gato();
misAnimales[2] = new Animal();

misAnimales[0].Hablar();
misAnimales[1].Hablar();
misAnimales[2].Hablar();
}

}
```

La salida de este programa es:

```
Guau!
Miauuu
Estoy comunicándome...

Estoy comunicándome...
Estoy comunicándome...
Estoy comunicándome...
```

La primera parte era de esperar: si creamos un perro, debería decir "Guau", un gato debería decir "Miau" y un animal genérico debería comunicarse. Eso es lo que se consigue con este fragmento:

```
Perro miPerro = new Perro();
Gato miGato = new Gato();
Animal miAnimal = new Animal();

miPerro.Hablar();
miGato.Hablar();
miAnimal.Hablar();
```

En cambio, si creamos un array de animales, no se comporta correctamente, a pesar de que después digamos que el primer elemento del array es un perro:

```
Animal[] misAnimales = new Animal[3];

misAnimales[0] = new Perro();
misAnimales[1] = new Gato();
misAnimales[2] = new Animal();

misAnimales[0].Hablar();
misAnimales[1].Hablar();
misAnimales[2].Hablar();
```

Es decir, como la clase base es "Animal", el primer elemento hace lo que corresponde a un Animal genérico (decir "Estoy comunicándome"), a pesar de que hayamos dicho que se trata de un Perro.

Generalmente, no será esto lo que queramos. Sería interesante no necesitar crear un array de perros y otros de gatos, sino poder crear un array de animales, y que contuviera animales de distintos tipos.

Para conseguir este comportamiento, debemos indicar a nuestro compilador que el método "Hablar" que se usa en la clase Animal puede que sea redefinido por otras clases hijas, y que en ese caso debe prevalecer lo que indiquen las clases hijas.

La forma de hacerlo es declarando ese método "Hablar" como "virtual", y empleando en las clases hijas la palabra "override" en vez de "new", así:

```
/*
 * Ejemplo en C# nº 66:
 * ejemplode66.cs
 */
/*
 * Ejemplo de clases
 * Array de objetos de
 * varias subclases con
 * metodos virtuales
 */
/*
 * Introducción a C#,
 * Nacho Cabanes
 */
using System;

public class Animal
{
    public virtual void Hablar()
    {
        Console.WriteLine("Estoy comunicándome...");
    }
}

public class Perro: Animal
{
    public override void Hablar()
    {
        Console.WriteLine("Guau!");
    }
}

public class Gato: Animal
{
```

```

public override void Hablar()
{
    Console.WriteLine("Miauuu");
}
// ----

public class Ejemplo66
{

    public static void Main()
    {

        // Primero creamos un animal de cada tipo
        Perro miPerro = new Perro();
        Gato miGato = new Gato();
        Animal miAnimal = new Animal();

        miPerro.Hablar();
        miGato.Hablar();
        miAnimal.Hablar();

        // Linea en blanco, por legibilidad
        Console.WriteLine();

        // Ahora los creamos desde un array
        Animal[] misAnimales = new Animal[3];

        misAnimales[0] = new Perro();
        misAnimales[1] = new Gato();
        misAnimales[2] = new Animal();

        misAnimales[0].Hablar();
        misAnimales[1].Hablar();
        misAnimales[2].Hablar();
    }
}

```

El resultado de este programa ya sí es el que posiblemente deseábamos: tenemos un array de animales, pero cada uno "Habla" como corresponde a su especie:

Guau!
Miauuu
Estoy comunicándome...

Guau!
Miauuu
Estoy comunicándome...

Ejercicio propuesto:

- **(7.3.1)** Crea una versión ampliada del ejercicio 7.2.1, en la que no se cree un único objeto de cada clase, sino un array de tres objetos.

Ejercicios propuestos:

- **(6.5.1)** Ampliar las clases del ejercicio 6.4.1, para que todas ellas contengan constructores. Los constructores de casi todas las clases estarán vacíos, excepto del de "Calefactor", que prefijará una temperatura de 25 grados.

6.6. Polimorfismo y sobrecarga

Esos dos constructores "Puerta()" y "Puerta(int ancho, int alto)", que se llaman igual pero reciben distintos parámetros, y se comportan de forma que puede ser distinta, son ejemplos de "**polimorfismo**" (funciones que tienen el mismo nombre, pero distintos parámetros, y que quizás no se comporten de igual forma).

Un concepto muy relacionado con el polimorfismo es el de "**sobrecarga**": dos funciones están sobrecargadas cuando se llaman igual, reciben el mismo número de parámetros, pero se aplican a objetos distintos, así:

```
puerta.Abrir ();
libro.Abrir ();
```

En este caso, la función "Abrir" está sobrecargada: se usa tanto para referirnos a abrir un libro como para abrir una puerta. Se trata de dos acciones que no son exactamente iguales, que se aplican a objetos distintos, pero que se llaman igual.

Ejercicios propuestos:

- **(6.6.1)** Añade a la clase "Ventana" un nuevo método Abrir, que reciba un parámetro, que será un número del 0 al 100 que indique hasta qué punto se debe abrir la ventana (100=100%, abierta; 0=0%, cerrada). Crea los atributos auxiliares que necesites para reflejar esa información.

6.7. Orden de llamada de los constructores

Cuando creamos objetos de una clase derivada, antes de llamar a su constructor se llama a los constructores de las clases base, empezando por la más general y terminando por la más específica. Por ejemplo, si creamos una clase "GatoSiamés", que deriva de una clase "Gato", que a su vez procede de una clase "Animal", el orden de ejecución de los constructores sería: Animal, Gato, GatoSiames, como se ve en este ejemplo:

```
/*
 * Ejemplo en C# nº 62:
 * ejemplo62.cs
 *
 * Cuarto ejemplo de clases
 * Constructores y herencia
 *
 * Introducción a C#
 * Nacho Cabanes
 */
```