3. Tipos de datos básicos

3.1. Tipo de datos entero

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

Empieza a ser el momento de refinar, de dar más detalles. El primer "matiz" importante que hemos esquivado hasta ahora es el "tamaño" de los números que podemos emplear en nuestros programas, así como su signo (positivo o negativo), y esos son detalles importantes cuando se emplean muchos de los lenguajes que derivan de C (como C#, C++ y Java). Por ejemplo, un dato de tipo "int" puede guardar números de hasta unas nueve cifras, tanto positivos como negativos, y ocupa 4 bytes en memoria. Por ello, los "int" no serán adecuados si necesitamos almacenar números de más de 10 cifras, ni si debemos emplear cifras decimales.

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

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

3.1.1. Tipos de datos para números enteros

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

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

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

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

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

using System;

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

Ejercicios propuestos:

(3.1.1.1) Calcula el producto de 1.000.000 por 1.000.000, usando una variable llamada "producto", de tipo "long". Prueba también a calcularlo usando una variable de tipo "int".

3.1.2. Conversiones de cadena a entero

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

Así, para datos de tipo "byte" usaremos Convert.ToByte (sin signo) y ToSByte (con signo), para datos de 2 bytes (short) tenemos ToInt16 (con signo) y ToUInt16 (sin signo), y para los de 8 bytes (long) existen ToInt64 (con signo) y ToUInt64 (sin signo). De igual modo, para los enteros de 32 bits sin signo se empleará ToUInt32.

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

using System;

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

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

Ejercicios propuestos:

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

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

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

3.1.3. Incremento y decremento

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

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

Pues bien, en C# (y en otros lenguajes que derivan de C, como C++, Java y PHP), existe una notación más compacta para esta operación, y para la opuesta (el decremento):

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

Un ejemplo básico de su uso es:

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

using System;

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

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

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

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

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

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

En cambio, si escribimos

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

y "a" valía 2, primero aumenta "a" y luego se asigna ese valor a "b" (**preincremento**), de modo que a=3 y b=3.

Un ejemplo más detallado:

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

using System;

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

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

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

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

Ejercicios propuestos:

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

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

3.1.4. Operaciones abreviadas: +=

Aún hay más. Tenemos incluso formas reducidas de escribir operaciones como "a = a+5". Estas son las abreviaturas más habituales:

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

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

using System;

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

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

Ejercicios propuestos:

(3.1.4.1) Crea un programa que use tres variables x,y,z. Sus valores iniciales serán 15, -10, 214. Deberás incrementar el valor de estas variables en 12, usando el formato abreviado. ¿Qué valores esperas que se obtengan? Contrástalo con el resultado obtenido por el programa.

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

3.1.5. Asignaciones múltiples

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

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

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

using System;

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

3.1.6. Operaciones con bits

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

Operación	Resultado	En C#	Ejemplo
Complemento (not)	Cambiar 0 por 1 y viceversa	~	~1100 = 0011
Producto lógico (and)	1 sólo si los 2 bits son 1	&	1101 & 1011 = 1001
Suma lógica (or)	1 sólo si uno de los bits es 1	|	1101 | 1011 = 1111
Suma exclusiva (xor)	1 sólo si los 2 bits son distintos	^	1101 ^ 1011 = 0110
Desplazamiento a la izquierda	Desplaza y rellena con ceros	<<	1101 << 2 = 110100
Desplazamiento a la derecha	Desplaza y rellena con ceros	>>	1101 >> 2 = 0011

Un ejemplo de su uso podría ser:

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

using System;

class Bits
{
    static void Main()
    {
        int a   = 67;
        int b   =  33;

        Console.WriteLine("La variable a vale {0}", a);
        Console.WriteLine("y b vale {0}", b);
        Console.WriteLine("  El complemento de a es: {0}", ~a);
        Console.WriteLine("  El producto lógico de a y b es: {0}", a&b);
        Console.WriteLine("  Su suma lógica es: {0}", a|b);
        Console.WriteLine("  Su suma lógica exclusiva es: {0}", a^b);
        Console.WriteLine("  Desplacemos a a la izquierda: {0}", a << 1);
        Console.WriteLine("  Desplacemos a a la derecha: {0}", a >> 1);
    }
}
```

El resultado es:

```
La variable a vale 67
y b vale 33
  El complemento de a es: -68
  El producto lógico de a y b es: 1
  Su suma lógica es: 99
  Su suma lógica exclusiva es: 98
  Desplacemos a a la izquierda: 134
  Desplacemos a a la derecha: 33
```

Para comprobar que es correcto, podemos convertir al sistema binario esos dos números y seguir las operaciones paso a paso:

```
67 = 0100 0011
33 = 0010 0001
```

En primer lugar complementamos "a", cambiando los ceros por unos:

```
1011 1100 = -68
```

Después hacemos el producto lógico de A y B, multiplicando cada bit, de modo que 1*1 = 1, 1*0 = 0, 0*0 = 0

```
0000 0001 = 1
```

Después hacemos su suma lógica, sumando cada bit, de modo que 1+1 = 1, 1+0 = 1, 0+0 = 0

```
0110 0011 = 99
```

La suma lógica exclusiva devuelve un 1 cuando los dos bits son distintos: 1^1 = 0, 1^0 = 1, 0^0 = 0

```
0110 0010 = 98
```

Desplazar los bits una posición a la izquierda es como multiplicar por dos:

```
1000 0110 = 134
```

Desplazar los bits una posición a la derecha equivale a dividir entre dos:

```
0010 0001 = 33
```

¿Qué utilidades puede tener todo esto? Posiblemente, más de las que parece a primera vista. Por ejemplo: desplazar a la izquierda es una forma muy rápida de multiplicar por potencias de dos; desplazar a la derecha equivale a dividir por potencias de dos; la suma lógica exclusiva (xor) es un método rápido, sencillo y reversible de cifrar mensajes; el producto lógico nos permite obligar a que ciertos bits sean 0 (algo que se puede usar para comprobar máscaras de red); la suma lógica, por el contrario, puede servir para obligar a que ciertos bits sean 1...

Un último comentario: igual que hacíamos operaciones abreviadas como

```
x += 2;
```

también podremos hacer cosas como

```
x <<= 2;
x &= 2;
x |= 2;
   ...
```

Ejercicios propuestos

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

3.2. Tipo de datos real

Cuando queremos almacenar datos con decimales, no nos sirve el tipo de datos "int". Necesitamos otro tipo de datos que sí esté preparado para guardar números "reales" (con decimales). Al igual que ocurría con los números enteros, tendremos más de un tipo de número real para elegir.

3.2.1. Coma fija y coma flotante

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

Coma fija: el número máximo de cifras decimales está fijado de antemano, y el número de cifras enteras también. Por ejemplo, con un formato de 3 cifras enteras y 4 cifras decimales, el número 3,75 se almacenaría correctamente (como 003,7500), el número 970,4361 también se guardaría sin problemas, pero el 5,678642 se guardaría como 5,6786 (se perdería a partir de la cuarta cifra decimal) y el 1020 no se podría guardar de forma correcta (tiene más de 3 cifras enteras). Esta forma de almacenar números reales no se suele emplear en los sistemas informáticos modernos.

Coma flotante: la cantidad de decimales y de cifras enteras permitida es variable, lo que importa es la cantidad de cifras significativas (a partir del último 0). Por ejemplo, con 5 cifras significativas se podrían almacenar números como el 13405000000 o como el 0,0000007349 pero no se guardaría correctamente el 12,0000034, que se redondearía a un número cercano.

Casi cualquier lenguaje de programación actual permite emplear números de coma flotante. En C# corresponden al tipo de datos llamado "float" (aunque hay más posibilidades, como veremos dentro de poco).

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

using System;

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

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

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

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

```
    }
}
```

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

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

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

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

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

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

Ejercicios propuestos:
(3.2.1.1) Crea un programa que muestre el resultado de dividir 3 entre 4, primero usando números enteros y luego usando números de coma flotante.
(3.2.1.2) ¿Cuál sería el resultado de las siguientes operaciones, usando números reales? a=5; a/=2; a+=1; a*=3; --a;

3.2.2. Simple y doble precisión

En la mayoría de lenguajes de programación, contamos con dos tamaños de números reales para elegir, según si queremos guardar números con mayor cantidad de cifras o con menos. Para números con pocas cifras significativas (un

máximo de 7, lo que se conoce como "un dato real de simple precisión") usaremos el tipo "float" y para números que necesiten más precisión (unas 15 cifras, "doble precisión") disponemos del tipo "double". En C# existe un tercer tipo de números reales, con mayor precisión todavía, el tipo "decimal", que se acerca a las 30 cifras significativas:

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

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

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

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

using System;

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

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

Ahora su resultado sería:

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

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

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

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

```
// Ejemplo_03_02_02b.cs
// Números reales: valor inicial de un float y de un double
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_02_02b
{
    static void Main()
    {
        int radio;
        float piFloat = 3.14159265358979238f;  // Atención a la "f" del final
        double piDouble = 3.14159265358979238; // Sin "f"

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

Y su resultado sería algo como

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

Ejercicios propuestos:

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

(3.2.2.2) Calcula el área de un círculo, dado su radio, que será un número entero (área = pi * radio al cuadrado). Usa datos de doble precisión.

3.2.3. Pedir números reales al usuario

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

```
// Ejemplo_03_02_03a.cs
// Números reales: pedir al usuario
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_02_03a
{
    static void Main()
    {
        float primerNumero;
        float segundoNumero;
        float suma;

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

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

Cuidado al probar este programa: en el fuente debemos escribir los decimales usando **un punto**, como 123.456, porque el lenguaje C# se apoya en construcciones del idioma inglés. Pero al poner el ejecutable en marcha, cuando se pidan datos al usuario, parte del trabajo se le encarga al sistema operativo, de modo que si éste sabe que en nuestro país se usa la "coma" para separar los decimales, considerará que la coma es el separador correcto y no el punto, que será ignorado. Por ejemplo, ocurre si introducimos los datos 23,6 y 34.2 en la versión española de Windows 10, donde obtendremos como respuesta:

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

Ejercicios propuestos:

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

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

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

(3.2.3.4) Si se ingresan E euros en el banco a un cierto interés I durante N años, el dinero obtenido viene dado por la fórmula del interés compuesto: Resultado = $e(1+ i)^n$ Aplícalo para calcular en cuanto se convierten 1.000 euros al cabo de 10 años al 3% de interés anual.

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

(3.2.3.6) Crea un programa que "dibuje" la gráfica de $y = (x-5)^2$ para valores de x entre 1 y 10. Deberá hacerlo dibujando varios espacios en pantalla y luego un asterisco. La cantidad de espacios dependerá del valor obtenido para "y". Te será fácil si dibujas la gráfica "girada", de forma que los valores de "y" crezcan hacia la derecha, así:

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

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

3.2.4. Conversión de tipos (typecast)

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

- Usar Convert, como en x = Convert.ToInt32(y);
- Hacer un **forzado de tipos**, que es más rápido pero no siempre es posible, sólo cuando el tipo de origen y el de destino se parecen lo suficiente (habitual entre dos datos numéricos, no tanto para convertir de número a

texto y viceversa). Para ello, se precede el valor de la variable con el nuevo tipo de datos entre paréntesis, así: x = (int) y;

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

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

using System;

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

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

        longitud = 2 * pi * radio;
        Console.WriteLine("La longitud de la circunferencia es");
        Console.WriteLine(longitud);

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

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

Su resultado sería:

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

Ejercicios propuestos:

(3.2.4.1) Crea un programa que calcule la raíz cuadrada del número que introduzca el usuario. La raíz se deberá calcular como "double", pero el resultado se mostrará como "float". (Recuerda: como viste al hacer el ejercicio 3.2.3.3, la raíz cuadrada de un número x se calcula con Math.Sqrt(x)).

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

3.2.5. Formatear números

En más de una ocasión nos interesará afinar la apariencia de los números en pantalla, para mostrar sólo una cierta cantidad de decimales: por ejemplo, nos puede interesar que una cifra que corresponde a dinero se muestre siempre con dos cifras decimales, o que una nota se muestre redondeada, sin decimales, o bien con sólo un decimal.

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

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

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

Vamos a probarlos en un ejemplo:

```
// Ejemplo_03_02_05a.cs
// Formato de números reales
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_02_05a
{
    static void Main()
```

```
    {
        double numero = 12.34;

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

El resultado de este ejemplo sería:

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

Como se puede ver, ocurre lo siguiente:

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

Ejercicios propuestos:

(3.2.5.1) El usuario de nuestro programa podrá teclear dos números de hasta 12 cifras significativas. El programa deberá mostrar el resultado de dividir el primer número entre el segundo, utilizando tres cifras decimales.

(3.2.5.2) Crea un programa que use tres variables x,y,z. Las tres serán números reales, y nos bastará con datos de simple precisión. Se deberá pedir al usuario los valores para las tres variables y mostrar en pantalla el valor de x^2 + y - z (con exactamente dos cifras decimales).

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

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

3.2.6. Cambios de base

Un uso alternativo de ToString es el de **cambiar un número de base**. Por ejemplo, habitualmente trabajamos con números decimales (en base 10), pero en informática son también muy frecuentes la base 2 (el sistema binario) y la base 16 (el sistema hexadecimal). Podemos convertir un número a binario o hexadecimal (o a base octal, menos frecuente) usando Convert.ToString e indicando la base, como en este ejemplo:

```
// Ejemplo_03_02_06a.cs
// De decimal a hexadecimal y binario
// Introducción a C#, por Nacho Cabanes

using System;

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

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

Su resultado sería:

```
f7
11110111
```

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

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

Ejercicios propuestos:

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

(3.2.6.2) Crea un programa que pida al usuario la cantidad de rojo (por ejemplo, 255), verde (por ejemplo, 160) y azul (por ejemplo, 0) que tiene un color, y que muestre ese color RGB en notación hexadecimal (por ejemplo, FFA000).
(3.2.6.3) Crea un programa para mostrar los números del 0 a 255 en hexadecimal, en 16 filas de 16 columnas cada una (la primera fila contendrá los números del 0 al 15 –decimal-, la segunda del 16 al 31 –decimal- y así sucesivamente).

Para convertir en sentido contrario, de **hexadecimal o binario a decimal**, podemos usar Convert.ToInt32, como se ve en el siguiente ejemplo. Es importante destacar que una constante hexadecimal se puede expresar precedida por "**0x**", como en "int n1 = 0x23;" (donde n1 tendría el valor 16*2 + 3*1 = 35, expresado en base 10). En los lenguajes C y C++, un valor precedido por "0" se considera **octal**, de modo que para "int n2 = 023;" el valor decimal de n2 sería 8*2 + 3*1 = 17, pero en C# no es así: un número que empiece por 0 (no por 0x) se considera que está escrito en base 10, como se ve en este ejemplo:

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

using System;

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

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

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

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

Que mostraría:

```
19 26 13 12 201
```

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

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

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

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

o incluso de combinar ambas posibilidades:

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

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

3.2.7. Funciones matemáticas

En C# disponemos de muchas funciones matemáticas predefinidas, como:

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

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

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

También tenemos una serie de constantes como

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

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

- La raíz cuadrada de 4 se calcularía haciendo `x = Math.Sqrt(4);`
- La potencia: para elevar 2 al cubo haríamos `y = Math.Pow(2, 3);`
- El valor absoluto: para trabajar sólo con números positivos usaríamos `n = Math.Abs(x);`

Un ejemplo más avanzado, usando funciones trigonométricas, que calculase el "coseno de 45 grados" podría ser:

```
// Ejemplo_03_02_07a.cs
// Ejemplo de funciones trigonométricas
// Introducción a C#, por Nacho Cabanes

using System;

public class Ejemplo_03_02_07a
{
    public static void Main()
    {
        double anguloGrados = 45;
        double anguloRadianes = anguloGrados * Math.PI / 180.0;

        Console.WriteLine("El coseno de 45 grados es: {0}",
            Math.Cos(anguloRadianes));
    }
}
```

Ejercicios propuestos:

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

(3.2.7.2) Diseña un programa que calcule cualquier raíz (de cualquier orden) de un número. El usuario deberá indicar el número (por ejemplo, 2) y el índice de la raíz (por ejemplo, 3 para la raíz cúbica). Pista: hallar la raíz cúbica de 2 es lo mismo que elevar 2 a 1/3.
(3.2.7.3) Crea un programa que calcule la distancia entre dos puntos (x_1,y_1) y (x_2,y_2), usando la expresión d = raíz [$(x_1-x_2)^2 + (y_1-y_2)^2$].
(3.2.7.4) Crea un programa que pida al usuario un ángulo (en grados) y muestre su seno, coseno y tangente. Recuerda que las funciones trigonométricas esperan que el ángulo se indique en radianes, no en grados. La equivalencia es que 180 grados = PI radianes.

3.3. Tipo de datos carácter

3.3.1. Leer y mostrar caracteres

Como ya vimos brevemente, en C# también tenemos un tipo de datos que nos permite almacenar una única letra, el tipo "char":

```
char letra;
```

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

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

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

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

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

```
// Ejemplo_03_03_01a.cs
// Tipo de datos "char"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_03_01a
{
    static void Main()
    {
        char letra;
```

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

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

Ejercicios propuestos

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

(3.3.1.2) Crea un programa que muestre letras alternas (una sí y una no) entre la que teclee el usuario y la "z". Por ejemplo, si el usuario introduce una "a", se escribirá "aceg...".

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

```
XXXX
XXXX
XXXX
```

3.3.2. Secuencias de escape: \n y otras

Como hemos visto, los textos que aparecen en pantalla se escriben con WriteLine, indicados entre paréntesis y entre comillas dobles. Entonces surge una dificultad: ¿cómo escribimos una comilla doble en pantalla? La forma de conseguirlo es usando ciertos caracteres especiales, lo que se conoce como "secuencias de escape". Estos caracteres especiales se preceden con una barra invertida (\). Por ejemplo, con \" se escribirán unas **comillas dobles**, con \' unas **comillas simples**, con \\ se escribe una barra invertida y con \n se avanzará a la línea siguiente de pantalla (es preferible evitar este último, y usar WriteLine como hemos hecho hasta ahora, porque \n puede no funcionar correctamente en todos los sistemas operativos; más adelante veremos una alternativa más segura).

Estas secuencias especiales son las siguientes:

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

Vamos a ver un ejemplo que use las más habituales:

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

using System;

class Ejemplo_03_03_02a
{
    static void Main()
    {
        Console.WriteLine("Esta es una frase");
        Console.WriteLine();
        Console.WriteLine();
        Console.WriteLine("y esta es otra, separada dos lineas");

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

Su resultado sería este:

```
Esta es una frase


y esta es otra, separada dos lineas


Juguemos mas:

otro salto
Comillas dobles: ", simples ', y barra \
```

En algunas ocasiones puede ser incómodo manipular estas secuencias de escape. Por ejemplo, cuando usemos estructuras de directorios al estilo de MsDos y Windows, deberíamos duplicar todas las barras invertidas: c:\\datos\\ ejemplos\\curso\\ejemplo1. En este caso, como alternativa, se puede usar una **arroba** (@) antes del texto (e incluso de las comillas), en vez de usar las barras invertidas:

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

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

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

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

3.4. Toma de contacto con las cadenas de texto

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

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

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

```
// Ejemplo_03_04a.cs
// Uso básico de "string"
// Introducción a C#, por Nacho Cabanes

using System;

class Ejemplo_03_04a
{
    static void Main()
    {
```

```
        string frase;

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

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

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

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

Ejercicios propuestos:

(3.4.1) Crea un programa que pida al usuario su nombre, y le diga "Hola" si se llama "Juan", o bien le diga "No te conozco" si teclea otro nombre.

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

3.5. Los valores "booleanos"

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

```
bool encontrado;
encontrado = true;
```

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

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

```
// Ejemplo básico
partidaTerminada = false;
```

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

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

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

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

using System;

class Ejemplo_03_05a
{
    static void Main()
    {
        char letra;
        bool esVocal, esCifra;

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

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

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

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

Ejercicios propuestos:

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

(3.5.2) Crea una versión alternativa del ejercicio 3.5.1, que use "if" en vez del operador condicional.
(3.5.3) Crea una versión alternativa del ejercicio 3.5.1, que asigne directamente el valor al booleano a partir de una comparación.
(3.5.4) Crea un programa que use el operador condicional para dar a una variable llamada "ambosPares" (booleana) el valor "true" si dos números introducidos por el usuario son pares, o "false" si alguno es impar.
(3.5.5) Crea una versión alternativa del ejercicio 3.5.4, que use "if" en vez del operador condicional.
(3.5.6) Crea una versión alternativa del ejercicio 3.5.5, que asigne directamente el valor al booleano a partir de una comparación.

3.6. Constantes y enumeraciones

En ocasiones, manejaremos valores que realmente no van a variar. Esto podría ocurrir, por ejemplo, con el número Pi, frecuente en matemáticas. En esos casos, por legibilidad y por facilidad de mantenimiento del programa, puede ser preferible no usar el valor numérico, sino una variable. Dado que el valor de ésta no debería cambiar, podemos usar la palabra "const" para indicar que debe ser constante, y así el compilador no permitirá que la modifiquemos más adelante por error. Por convenio, para que sea fácil distinguir una constante de una variable, se suele escribir su nombre totalmente en mayúsculas:

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

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

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

using System;

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

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

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

Ejercicios propuestos

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

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

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

Pero también existe una forma alternativa de hacerlo, especialmente útil si son números enteros consecutivos. Se trata de **enumerarlos**:

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

(Al igual que las constantes de cualquier otro tipo, se puede escribir en mayúsculas para recordar "de un vistazo" que son constantes, no variables)

La primera constante valdrá 0, y las demás irán aumentando de una en una, de modo que en nuestro caso valen:

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

Si queremos que los valores no sean exactamente estos, podemos dar valor a cualquiera de las contantes, y las siguientes irán aumentando de uno en uno. Por ejemplo, si escribimos

```
    enum diasSemana { LUNES=1, MARTES, MIERCOLES, JUEVES=6, VIERNES,
      SABADO=10, DOMINGO };
```

Ahora sus valores son:

```
   LUNES = 1, MARTES = 2, MIERCOLES = 3, JUEVES = 6, VIERNES = 7,
   SABADO = 10, DOMINGO = 11
```

Un ejemplo básico podría ser

```
// Ejemplo_03_06b.cs
// Ejemplo de enumeraciones
// Introducción a C#, por Nacho Cabanes
```

```
using System;

class Enumeraciones
{

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

    static void Main()
    {
        Console.Write("En la enumeracion, el miércoles tiene el valor: {0} ",
            diasSemana.MIERCOLES);
        Console.WriteLine("que equivale a: {0}",
            (int) diasSemana.MIERCOLES);

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

        Console.WriteLine("En las constantes, tiene el valor: {0}",
            MIERCOLES);
    }
}
```

y su resultado será:

```
En la enumeracion, el miércoles tiene el valor: MIERCOLES que equivale a: 2
En las constantes, tiene el valor: 2
```

Nota 1: las enumeraciones existen también en otros lenguajes como C y C++, pero la sintaxis es ligeramente distinta: en C# es necesario indicar el nombre de la enumeración cada vez que se utilicen sus valores (como en diasSemana.MIERCOLES), mientras que en C se usa sólo el valor (MIERCOLES).

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

Ejercicios propuestos

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

3.7. Variables con tipo implicito

A partir de Visual C# 3.0, existe la posibilidad de no declarar de forma explícita el tipo de una variable, sino que sea el propio compilador el que lo deduzca del contexto. Para ello, se utiliza la palabra "var", como en este ejemplo:

```
// Ejemplo_03_07a.cs
// Ejemplo de uso de "var"
// Introducción a C#, por Nacho Cabanes

using System;

class UsoVar
{
    static void Main()
    {
        var n = 5;
        Console.WriteLine("n vale {0} y es de tipo {1}",
            n, n.GetType());

        var condicion = 5 == 7;
        Console.WriteLine("condicion vale {0} y es de tipo {1}",
            condicion, condicion.GetType());

        var letra = 'a';
        Console.WriteLine("letra vale {0} y es de tipo {1}",
            letra, letra.GetType());

        var pi = 3.1416;
        Console.WriteLine("pi vale {0} y es de tipo {1}",
            pi, pi.GetType());

        var texto = "Hola";
        Console.WriteLine("texto vale {0} y es de tipo {1}",
            texto, texto.GetType());
    }
}
```

Como se ve en este ejemplo, si necesitáramos saber de qué tipo es una variable (lo que no es habitual, porque si se usa "var" es para despreocuparnos de esos detalles), lo podríamos conseguir con "GetType()".

Ejercicios propuestos

(3.7.1) Crea un programa que pida al usuario una cantidad de kilómetros y muestre su equivalencia en millas. El valor de conversión debe estar en una variable definida con "var".

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