GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOi Escuela de organización industrial

CURSO DE PROGRAMACIÓN.NET

M.374.001.003

FUNCIONES

ENUMERACIONES

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

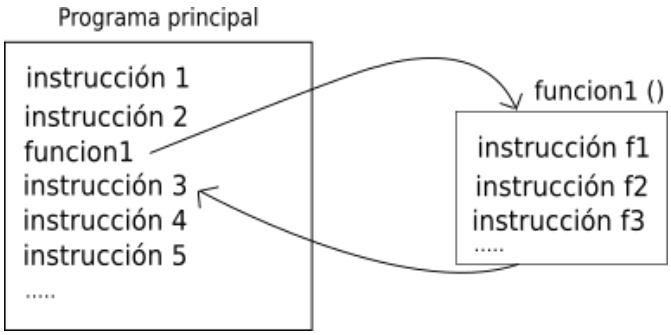
Ayuntamiento de Málaga

polo de contenidos digitales

Index

Funciones / métodos

Una función es un bloque de código asociado a un identificador (como los nombres de las variables), que puede recibir parámetros de entrada y devolver valores. Cada vez que invoquemos a la función se ejecutará el código interno de la misma. Una vez acaba de ejecutar, el flujo de programa se sitúa en la línea siguiente a la llamada que hicimos. Las funciones se crean para aislar código reutilizable y no tener que repetir dicho código cada vez que queramos hacer una determinada operación (por ejemplo, una operación matemática compleja), sino simplemente llamar a la función y que esta nos devuelva el resultado.



En muchos lenguajes orientados a objetos como C# o Java no existe el concepto de función aislada como puede suceder en lenguajes estructurados como C. Es decir, las funciones siempre estarán asociadas a una clase determinada. Este tipo de funciones asociadas a una clase reciben el nombre de métodos, y ya los estudiaremos en más profundidad en el bloque de programación orientada a objetos.

Sin embargo, es posible hacer una equivalencia utilizando métodos estáticos. Para ello, antes de la función debemos poner las palabras clave (que más adelante entenderemos correctamente lo que significan) **public static**. Estos métodos los crearemos dentro de una clase, que puede ser la clase principal, y no podemos situar un método dentro de otro. De hecho, ya tenemos un método con estas características creado, que es el método **Main**, que define el punto de entrada al programa principal.

Cabe destacar que en C# los nombres de los métodos se escriben empezando por mayúscula, cosa que no ocurre en la mayoría de lenguajes. Vamos a ver como crear un método estático (equivalente a una función) básico que simplemente nos salude y como llamarlo.

```
class Program
{
    // Método (función) que nos saluda
    public static void Saluda()
    {
        Console.WriteLine("¡Hola!. Soy una función.");
    }

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

```
    {
        // Vamos a llamarlo 2 veces
        Saluda();
        Saluda();
    }
}
```

```
¡Hola!. Soy una función.
¡Hola!. Soy una función.
```

Observa que hemos añadido también la palabra **void** delante del nombre. Esto indica que no va a devolver ningún valor como respuesta. Los paréntesis vacíos significan que tampoco va a recibir parámetros o valores de entrada.

Parámetros de entrada

Al crear la función, podemos establecer unos datos de entrada que dicha función deberá recibir antes de ejecutar su código. Al igual que las variables, debemos indicar el tipo de cada parámetro, y estos se comportarán como variables locales a la función. Es decir, solo existirán dentro de esta.

```
class Program
{
    public static void Saluda (string nombre)
    {
        Console.WriteLine($"Hola {nombre}");
    }

    static void Main(string[] args)
    {
        Saluda("Pepe"); // Hola Pepe
        Saluda("Marta"); // Hola Marta
    }
}
```

```
class Program
{
    public static void Suma(int n1, int n2)
    {
         Console.WriteLine($"{n1} + {n2} = {n1 + n2}");
    }

    static void Main(string[] args)
    {
         Suma(2, 5); // 2 + 5 = 7
         Suma(10, 20); // 10 + 20 = 30
    }
}
```

Parámetros opcionales

Existe la posibilidad de establecer valores por defecto a los parámetros de una función, de manera que no sea necesario pasárselos si queremos utilizar estos valores dentro de la función. Esto convierte a dichos parámetros en opcionales.

```
class Program
```

```
{
    public static void Saluda(string nombre = "Anónimo")
    {
        Console.WriteLine($"Hola {nombre}");
    }

    static void Main(string[] args)
    {
        Saluda("Pepe"); // Hola Pepe

        Saluda(); // Hola Anónimo

    }
}
```

Envío de parámetros por nombre

Al llamar a una función, normalmente debemos respetar el orden de los parámetros. Sin embargo, en C#, podemos usar el orden que queramos siempre y cuando especifiquemos el nombre del parámetro (**parámetro: valor**).

```
class Program
{
    public static void MuestraProducto(string nombre, double precio)
    {
        Console.WriteLine($"{nombre}: {precio:N2}");
    }

    static void Main(string[] args)
    {
        MuestraProducto(precio: 23.95, nombre: "Silla"); // Silla: 23,95
    }
}
```

Esta funcionalidad es útil cuando tenemos varios parámetros con valores por defecto, y queremos asignar valores a unos y dejar otros (que son anteriores) con su valor por defecto.

```
class Program
{
    public static void MuestraProducto(string nombre = "Prueba",
                                        double precio = 0.0)
    {
        Console.WriteLine($"{nombre}: {precio:N2}");
    }

    static void Main(string[] args)
    {
        MuestraProducto(precio: 23.95); // Prueba: 23,95

    }
}
```

Agrupación de parámetros (params)

Existe la posibilidad de llamar a una función con un número variable de parámetros. La única restricción es que todos esos parámetros deben ser del

mismo tipo. Al declarar la función, podemos utilizar la palabra **params** delante del último parámetro. Automáticamente, todos los parámetros que se le envíen a la función a partir de esa posición, se recogen en el último parámetro con dicha palabra clave. Eso sí, el parámetro debe ser un array para poder recoger más de un posible valor (o ninguno).

```
class Program
{
    public static void Suma(params int[] nums)
    {
        System.Console.WriteLine($"He recibido {nums.Length} números");
        int total = 0;
        for (int i = 0; i < nums.Length; i++)
        {
            total += nums[i];
        }
        System.Console.WriteLine($"La suma total es {total}");
    }

    static void Main(string[] args)
    {
        Suma();
        Suma(3, 5);
        Suma(3, 4, 8, 2);
    }
}
```

```
He recibido 0 números
La suma total es 0
He recibido 2 números
La suma total es 8
He recibido 4 números
La suma total es 17
```

Retorno de valores

Las funciones pueden devolver valores (por ejemplo, el resultado de una operación). Para establecer que una función devuelve un valor, debemos indicar el tipo de dato que devuelve justo antes del nombre de la misma (void significa que no devuelve nada).

```
class Program
{
    public static int Suma(params int[] nums)
    {
        int total = 0;
        for (int i = 0; i < nums.Length; i++)
        {
            total += nums[i];
        }
        return total;
    }

    static void Main(string[] args)
    {
        int resultado = Suma(3, 4, 8, 2);
        Console.WriteLine($"El resultado es {resultado}"); // El resultado es 17
    }
```

```
}
```

Parámetros por referencia (ref)

Cuando le pasamos una variable a una función como parámetro, se genera una copia del valor y se le asigna al parámetro. Esto es así para los tipos primitivos de datos (incluyendo string).

```
class Program
{
    public static void CambiaValor(int num)
    {
        // El parámetro es una copia de la variable enviada
        // No estamos modificando la variable original
        num = 13;
    }

    static void Main(string[] args)
    {
        int num = 1;
        CambiaValor(num);
        Console.WriteLine(num); // 1 (no ha cambiado)
    }
}
```

Si incluimos la palabra ref, en lugar del valor le estamos pasando una referencia a la variable que estamos usando en la llamada. Esto implica que el parámetro será equivalente a la variable externa a la función en todos los aspectos, y si modificamos su valor dentro, también se modifica fuera. Esta característica es peligrosa y puede tener efectos colaterales no deseados. Lenguajes como Java no la incluyen por eso.

```
class Program
{
    public static void CambiaValor(ref int num)
    {
        // El parámetro apunta a la variable original enviada
        // Las modificaciones aquí tienen efecto fuera
        num = 13;
    }

    static void Main(string[] args)
    {
        int num = 1;
        CambiaValor(ref num);
        Console.WriteLine(num); // 13 (ha cambiado...)
    }
}
```

En el caso de arrays y otras estructuras más avanzadas como objetos, lo que se pasa (copia) es la dirección de memoria donde está el dato. No se copia la estructura en si. Por lo que cualquier modificación a una posición del array o propiedad de un objeto dentro de la función se mantienen una vez esta termina. Es por ello que hay que llevar cuidado con lo que modificamos dentro de una función.

```
class Program
```

```
{
    public static void CambiaValor(int[] nums)
    {
        // El parámetro apunta al mismo dato en memoria que la variable enviada
        nums[1] = 99;
    }

    static void Main(string[] args)
    {
        int[] array = {1, 2, 3, 4};
        CambiaValor(array);
        Console.WriteLine(String.Join(',', array)); // 1,99,3,4
    }
}
```

En realidad estamos pasando el parámetro por copia (no usamos **ref**), pero en este caso como ya hemos dicho se está copiando la dirección de memoria (número entero) que apunta al array o al objeto, por lo que es similar a usar ref con una variable de tipo primitivo, con la salvedad de que si hacemos que el parámetro apunte a un nuevo array u objeto dentro de la función antes de modificar, no pasaría nada.

```
class Program
{
    public static void CambiaValor(int[] nums)
    {
        // Si apuntamos el parámetro a otro array (diferente dirección de memoria)
        // ya no tiene efectos sobre lo de fuera (si ponemos ref entonces sí...)
        nums = new int[] {10, 3, 4, 5};
    }

    static void Main(string[] args)
    {
        int[] array = {1, 2, 3, 4};
        CambiaValor(array);
        Console.WriteLine(String.Join(',', array)); // 1,2,3,4
    }
}
```

Una solución para evitar efectos colaterales sería generar una copia antes de enviársela a la función.

```
class Program
{
    public static void CambiaValor(int[] nums)
    {
        nums[1] = 99;
    }

    static void Main(string[] args)
    {
        int[] array = {1, 2, 3, 4};
        CambiaValor((int[])array.Clone()); // Evitamos modificación en el original
        Console.WriteLine(String.Join(',', array)); // 1,2,3,4
    }
}
```

Parámetros de salida (out)

La palabra **out** es similar a ref, solo que no requiere que la variable haya sido inicializada. El propósito es crear una función que pueda devolver más de un valor (el valor principal con return y el resto en los parámetros).

Cuando invoquemos a la función, en la posición de estos parámetros enviaremos una variable donde posteriormente podremos recoger los demás resultados devueltos.

```
class Program
{
    // El siguiente método devuelve un booleano indicando si ha encontrado
    // un valor en un array. Pero también devuelve la posición donde lo encontró
    public static bool BuscaValor(int[] arrayBuscar, int valor,
                                  out int posicion)
    {
        posicion = -1; // Por si no lo encuentra

        for(int i = 0; i < arrayBuscar.Length; i++)
        {
            if(arrayBuscar[i] == valor)
            {
                posicion = i;
                return true;
            }
        }

        return false;
    }

    static void Main(string[] args)
    {
        int[] array = {5, 12, 42, 15, 8};
        int posicion;
        bool encontrado = BuscaValor(array, 15, out posicion);
        if (encontrado)
        {
            System.Console.WriteLine($"Encontrado el valor 15 en la posición
{posicion}");
        }
    }
}
```

```
Encontrado el valor 15 en la posición 3
```

Funciones / métodos predefinidos en c#

Existen bastantes métodos ya implementados en C# para realizar operaciones con números, cadenas, arrays, fechas, etc. Vamos a ver algunos de ellos.

Funciones matemáticas

Existen varios métodos estáticos dentro de la clase Math para realizar operaciones matemáticas. Algunos de ellos son:

- **Math.Abs(número)** → Te devuelve el valor absoluto del número (sin signo).
- **Math.Round(número decimal)** → Redondea el número al entero más cercano. También podemos usar **Math.Round(número, decimales)** pasándole como segundo parámetro el número de decimales.
- **Math.Max(n1, n2)** y **Math.Min(n1, n2)** → Devuelve el máximo / mínimo entre 2 números.
- **Math.Sqrt(número)** → Devuelve la raíz cuadrada.
- **Math.Pow(n1, n2)** → Devuelve el primer número elevado al segundo.

```
double n1 = -23.40;
double n2 = 2.349532;

Console.WriteLine($"{Math.Abs(n1)}"); // 23,4
Console.WriteLine($"{Math.Round(n2, 2)}"); // 2,35
Console.WriteLine($"{Math.Max(23, 43)}"); // 43
Console.WriteLine($"{Math.Min(4, 5)}"); // 4
Console.WriteLine($"{Math.Sqrt(25)}"); // 5
```

Las clases que representan tipos numéricos primitivos como por ejemplo **Int32** (int), **Int16** (short), **Int64** (long), **Double**, etc. tienen métodos y propiedades que devuelven el número máximo o mínimo representable, si el número es negativo, infinito, etc.

```
short num = Int16.MaxValue;
Console.WriteLine(num); // 32767
Console.WriteLine(Double.IsNegative(-23)); // True
```

Funciones para trabajar con cadenas

La clase String también dispone de varios métodos predefinidos para operar con cadenas. Algunos de esos métodos son estáticos y otros no (ya veremos esos conceptos en profundidad). Por ahora vamos a ver algunos de ellos y como utilizarlos.

- **String.Concat(string, string)** → Devuelve las 2 cadenas recibidas concatenadas. Se le pueden pasar hasta 4 cadenas, o un array de string para concatenar.

```
string s1 = "abc";
string s2 = "def";
```

```
string s3 = "ghi";
string concatenada = String.Concat(s1, s2, s3);
Console.WriteLine(concatenada); // abcdefghi
```

- **String.Join(separador, array)** → Transforma un array a cadena de texto, separando los elementos por el separador que le pasemos.

```
double[] notas = { 5.65, 7.85, 6.5, 9 };
Console.WriteLine($"Mis notas son: {String.Join(", ", notas)}");

Mis notas son: 5,65, 7,85, 6,5, 9
```

- **String.Compare(cad1, cad2)** → Compara 2 cadenas. Devuelve un número negativo si la primera es menor (precede a la segunda en el orden), positivo si la segunda cadena va primero, y cero si son iguales.

```
string s1 = "a";
string s2 = "d";
Console.WriteLine(String.Compare(s1, s2)); // -1
```

- **cadena.IndexOf(subcadena)** → Busca una subcadena (también puede ser un char) dentro de la cadena y devuelve la primera posición (empezando por cero) donde se encuentra, o -1 si no la encuentra. Se puede enviar un segundo parámetro indicando a partir de qué posición buscar. **LastIndexOf** hace lo mismo empezando por el final.

```
string s = "Mi perro se llama Comeniños";
Console.WriteLine(s.IndexOf("perro")); // 3
Console.WriteLine(s.IndexOf("gato")); // -1
```

- **cadena.Contains(subcadena)** → Devuelve un booleano indicando si la subcadena está presente en la cadena (True) o no (False). Tenemos los métodos equivalentes **cadena.StartsWith** (comprueba si la cadena empieza por la subcadena) y **cadena.EndsWith** (comprueba si termina).

```
string s = "Mi perro se llama Comeniños";
Console.WriteLine(s.Contains("perro")); // True
Console.WriteLine(s.Contains("gato")); // False
```

- **cadena.Remove(inicio, numCar)** → Devuelve una nueva cadena borrando el número de caracteres indicado desde la posición "inicio". Si solo le pasamos la posición, borra desde ahí hasta el final de la cadena.
- **cadena.Insert(posición, cadena2)** → Devuelve una nueva cadena resultado de insertar cadena2 dentro de la cadena original en la posición indicada.

```
string s = "Mi perro se llama Comeniños";
string s1 = s.Remove(3, 5);
string s2 = s1.Insert(3, "koala");
Console.WriteLine(s2); // Mi koala se llama Comeniños
```

- **cadena.Replace(cadBuscar, cadSus)** → Devuelve una nueva cadena sustituyendo todas las ocurrencias de cadBuscar en la cadena por cadSus.

```
string s = "Mi perro se llama Comeniños";
string s1 = s.Replace("perro", "koala");
Console.WriteLine(s1); // Mi koala se llama Comeniños
```

- **cadena.Substring(inicio, numCar)** → Devuelve el trozo de cadena comprendido entre la posición de inicio y el número de caracteres indicado. Si no se indica número de caracteres, se devuelve hasta el final.

```
string s = "Mi perro se llama Comeniños";
string animal = s.Substring(3, 5);
Console.WriteLine(animal); // perro
```

- **cadena.ToLower** → Devuelve una nueva cadena transformando todas las letras de la cadena a minúsculas. El equivalente para transformar a mayúsculas es **cadena.ToUpper**.

```
string s = "HOLAAAAA";
Console.WriteLine(s.ToLower()); // holaaaaa
```

- **cadena.PadLeft(numCar)** → Siempre que el número de caracteres que le pasemos al método sea mayor que la longitud de la cadena, devuelve otra cadena de la longitud que le pasemos con espacios en blanco a la izquierda (alineada a la derecha). Para alinear la cadena a la izquierda usaríamos **cadena.PadRight**.

```
string s1 = "perro";
string s2 = "albaricoque";
string s3 = "ave";
Console.WriteLine(s1.PadLeft(14));
Console.WriteLine(s2.PadLeft(14));
Console.WriteLine(s3.PadLeft(14));
```

```
         perro
   albaricoque
           ave
```

- **cadena.Split(separador)** → A partir de una cadena, la trocea en función del separador que enviemos y devuelve un array de cadenas con los trozos generados.

```
string s = "pato-gato-perro-koala";
string[] animales = s.Split("-");
Console.WriteLine($"Hay {animales.Length} animales"); // Hay 4 animales
```

Funciones para trabajar con arrays

De forma similar a como ocurre con las cadenas, la clase Array de C# tiene una serie de métodos muy útiles para realizar operaciones con arrays como buscar, ordenar, etc. Algunos de ellos son:

- **Array.Fill(array, valor)** → Asigna una copia del valor recibido en todas las posiciones del array.

```
int[] nums = new int[10];
Array.Fill(nums, 1);
Console.WriteLine(String.Join(", ", nums)); // 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

- **Array.IndexOf(array, valor)** → Devuelve la posición del valor recibido en el array o -1 si no se encuentra. Usando LastIndexOf, la búsqueda empieza por el final.

```
int[] nums = {2, 4, 8, 12, 24};
Console.WriteLine(Array.IndexOf(nums, 8)); // 2
```

- **Array.Reverse(array)** → Modifica el array invirtiendo el orden de sus elementos (le da la vuelta).

```
int[] nums = {2, 4, 8, 12, 24};
Array.Reverse(nums);
Console.WriteLine(String.Join(", ", nums)); // 24, 12, 8, 4, 2
```

- **Array.Resize()** → Cambia el tamaño del array. Si lo hacemos más pequeño se eliminan las últimas posiciones.

```
int[] nums = {2, 4, 8, 12, 24};
Array.Resize(ref nums, 3);
Console.WriteLine(String.Join(", ", nums)); // 2, 4, 8
```

- **Array.Sort()** → Ordena el array. Si no se trata de un valor numérico o una cadena hay que pasarle una función de ordenación. Dicha función recibe 2 valores y al igual que String.Compare debe devolver un número negativo si el primero va delante (menor), positivo si el segundo va delante, o cero si son iguales.

```
int[] nums = {2, 45, 65, 36, 23};
Array.Sort(nums);
Console.WriteLine(String.Join(", ", nums)); // 2, 23, 36, 45, 65

string[] palabras = {"cerebro", "vaca", "si", "ele"};
// Ordenamos en función de la longitud de las palabras
Array.Sort(palabras, (p1, p2) => p1.Length.CompareTo(p2.Length));
Console.WriteLine(String.Join(", ", palabras)); // si, ele, vaca, cerebro
```

- **Array.Find(expresión)** → Devuelve el primer valor que cumple con la expresión recibida. Más adelante veremos como trabajar con expresiones. También tenemos FindLast (último valor), FindIndex (índice del primer valor), FindLastIndex (índice del último valor) y FindAll (devuelve un array con los elementos que cumplen la expresión).

```
int[] nums = {3, 23, 6, 93, 9, 12};
int[] numsMenor10 = Array.FindAll(nums, n => n < 10);
Console.WriteLine(String.Join(", ", numsMenor10)); // 3, 6, 9
```

Funciones para trabajar con fechas

Para trabajar con fechas, existe una clase específica en C# llamada DateTime que representa a una fecha, con una serie de métodos útiles para añadir o quitar días, meses, etc., o para imprimir la fecha en un determinado formato, por ejemplo.

Primero vamos a ver como crearíamos la siguiente fecha: 12 de agosto de 2019 a las 12:30:00 (el orden es siempre año, mes, día, hora, minuto, segundo).

```
DateTime fecha = new DateTime(2019, 8, 12, 12, 30, 0);
Console.WriteLine(fecha); // 12/8/19 12:30:00
```

Algunas propiedades y métodos interesantes para trabajar con fechas son:

- **DateTime.Now** → Devuelve la fecha actual, en el momento de la ejecución.

```
DateTime fecha = DateTime.Now;
Console.WriteLine(fecha); // 3/7/19 21:57:14
```

- **DateTime.Parse(fecha_string)** → Convierte una cadena que representa una fecha en un formato válido en un objeto del tipo fecha (DateTime). Si estamos utilizando un formato especial deberíamos usar **DateTime.ParseExact** e indicarle el tipo de formato.

```
string fechaStr = "31/12/2018";
DateTime fecha = DateTime.Parse(fechaStr);
Console.WriteLine(fecha); // 9/12/18 0:00:00

string fechaStr = "31-12-2018";
DateTime fecha = DateTime.ParseExact(fechaStr, "dd-mm-yyyy", // Formato
            System.Globalization.CultureInfo.InvariantCulture);
Console.WriteLine(fecha); // 31/12/18 0:00:00
```

Existe una forma más sencilla de hacerlo:

- **Conver.toDateTime(fecha_string)** → Convierte una cadena que representa una fecha en un formato válido en un valor DateTime.

```
string fechaStr1 = "31/12/2018";
DateTime fecha1 = Convert.ToDateTime(fechaStr1);
Console.WriteLine(fecha1); // 31/12/18 0:00:00

string fechaStr2 = "31-12-2018";
DateTime fecha2 = Convert.ToDateTime(fechaStr2);
Console.WriteLine(fecha2); // 31/12/18 0:00:00
```

- **fecha.Day, fecha.DayOfWeek, fecha.Month, fecha.Year**, etc. → Propiedades que te devuelven cada uno de los componentes de la fecha.

```
string[] dias = {"Domingo", "Lunes", "Martes", "Miércoles",
                 "Jueves", "Viernes", "Sábado"};
DateTime fecha = DateTime.Now;
string diaHoy = dias[(int)fecha.DayOfWeek];
Console.WriteLine($"Hoy es {diaHoy}, {fecha.Day:D2}/{fecha.Month:D2}/
{fecha.Year}");

Hoy es Miércoles, 03/07/2019
```

- **fecha.AddDays(num), fecha.AddMonths(num)**, etc. → Devuelve una nueva fecha añadiendo (o quitando si el número es negativo) un número de días, meses, años, horas, etc. a la fecha.

```
DateTime hoy = DateTime.Now;
DateTime ayer = hoy.AddDays(-1); // Ayer
```

Enumeraciones

Hay ocasiones en las que vamos a trabajar con una serie de valores acotados como pueden ser los días del mes, días de la semana, roles de usuario, etc. Si almacenamos dichos valores como un string o con un número que haga de equivalencia, corremos el riesgo de poder almacenar valores fuera de rango. Por ejemplo, si el mes lo almacenamos como entero, podría darse el caso de que tengamos un número mayor que 12 almacenado en la variable (obviamente intentaríamos que eso no pasara).

Para delimitar un conjunto de valores que puede tomar una variable o parámetro, suele ser conveniente utilizar enumeraciones. Una enumeración comienza por la palabra clave **enum** seguida de un nombre (que al igual que el de un método suele empezar por mayúscula en C#) que identifica la enumeración y la lista de valores entre llaves.

```
enum Dias {Lun, Mar, Mie, Jue, Vie, Sab, Dom};
```

Los valores por defecto tienen una equivalencia numérica ordenada empezando por cero. Es decir, Dias.Lun sería cero, Dias.Mar = 1, y así sucesivamente.

```
Console.WriteLine(Dias.Mie); // 2
```

Podemos crear variables de este tipo y usar todo tipo de comparaciones de tipo numérico.

```
Console.WriteLine(Dias.Mie); // 2

Dias dia = Dias.Sab;
if (dia > Dias.Jue)
{
    Console.WriteLine("Es fin de semana!");
}
```

Hay que tener en cuenta que aunque internamente estamos trabajando con valores numéricos, no se pueden hacer asignaciones directamente sin hacer una conversión de tipo previa.

```
int numDia = (int)dia; // Almacena 5 (sábado)
Dias dia2 = (Dias)(numDia + 1); // Dia.Dom (5+1)
```

Existe la posibilidad de iniciar el primer valor a un número diferente de cero. El resto de valores serán números consecutivos.

```
// Mar = 2, Mie = 3, ...
enum Dias {Lun = 1, Mar, Mie, Jue, Vie, Sab, Dom};
```

En realidad se le pueden asignar los valores numéricos que queramos a los elementos de una enumeración.

```
enum Puntuaciones {Baja=20, Media=50, Alta=70}
```