

# Unidad 9

Descargar estos apuntes en [pdf](#) o [html](#)

## Índice

- [Índice](#)
- ▼ [Tipos Enumerados](#)
  - [Introducción](#)
  - [Sintaxis](#)
  - [Conversiones con enumeraciones](#)
  - [Métodos de utilidad para enumeraciones](#)
  - [Enumeraciones NO excluyentes \(Máscara de Bits de estado o Flags\)](#)

# Tipos Enumerados

## Introducción

- Internamente se gestionan como objetos de tipo entero. Por tanto, son **tipos valor** y esto significa que en las asignaciones haremos una copia de su valor.
- Son útiles para **auto-documentar el código** y evitar **números mágicos**.
- Los utilizaremos siempre que queramos definir un conjunto finito de objetos o estados, **en lugar de definir constantes numéricas**.
- Solo podrán tomar valores, **mútualemente excluyentes**, dentro del rango definido, por lo que nos evitará errores derivados de valores inesperados.

## Sintaxis

- El identificador del tipo se escribirá en **PascalCasing** y debería estar en singular.
- Los identificadores de la enumeración se escribirán en **PascalCasing**.
- El si no lo especificamos por defecto es un **int** aunque podemos especificar otro tipos base enteros como: **byte** , **ushort** , etc.
- Para acceder a los valores pondremos: **NombreDelEnum.Identificador**

```
enum <NombreEnumeración> : <tipoBase>
{
    <Identificadores que definen el conjunto enumerado por extensión>
}
```

### Ejemplos:

```
enum Tamaño
{
    Pequeño, Mediano, Grande
}

Tamaño tamaño = default; // Equivale a hacer tamaño Pequeño
tamaño = Tamaño.Grande;
```

```

enum EstadoOrdenador
{
    Encendido, Apagado, Suspendido, Hibernado
}

enum Estación
{
    Primavera, Verano, Otoño, Invierno
}

```

Si no se especifica valor inicial para cada constante, el compilador les dará por defecto valores que empiecen desde 0 y se incrementen en una unidad para cada constante, según su orden de aparición en la definición de la enumeración. Así, el ejemplo del principio del tema es equivalente a escribir:

```

enum Tamaño : int
{
    Pequeño = 0, Mediano = 1, Grande = 2
}

```

Es posible **modificar el tipo base entero** y los valores iniciales de cada constante indicándolos explícitamente, como en el código recién mostrado. Otra posibilidad es alterar el valor base a partir del cual se va calculando el valor de las siguientes constantes, como en este otro ejemplo:

```

enum Tamaño : ushort
{
    Pequeño, Mediano = 5, Grande
}

```

En este último ejemplo mis enumerados ocuparán menos espacio en memoria por ser entero subyacente `ushort`. El valor asociado a `Pequeño` será **0**, el asociado a `Mediano` será **5**, y el asociado a `Grande` será 6, ya que como no se le indica explícitamente ningún otro, se considera que este valor es el de la constante anterior más 1.

Se puede especificarse el valor de un identificador en función del valor de otros como muestra este ejemplo:

```
enum Tamaño
{
    Pequeño, Mediano, Grande = Pequeño + Mediano
}
```

## Conversiones con enumeraciones

### 1. `enumerado.ToString()`

Pasa un enum a cadena.

### 2. `Enum.Parse(Type tipoDelEnum, string id, bool ignoraMayúsculas)`

```
Enum.Parse(Type tipoDelEnum, string id)
```

Pasa una cadena a enum.

### 3. `bool Enum.TryParse(string? id, , bool ignoraMayúsculas, out <MiTipoEnum> valorDelEnum)`

Intenta asociar una cadena a uno de los id definidos en el enum. Si lo consigue devuelve `true` y el enum a través de `valorDelEnum`.

### Ejemplos:

En los siguientes ejemplos vamos a ver cómo convertir entre un enum y una cadena, entre un enum y un entero, y viceversa...

Supongamos que tenemos el siguiente enum con los días de la semana:

```
public enum DiaSemana
{
    Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo
}
```

### Caso 1: Pasar de `enum` a `cadena`

```
DiaSemana dia = DiaSemana.Domingo;
string textoDia = dia.ToString();
Console.WriteLine(textoDia);

Console.WriteLine(DiaSemana.Martes);
```

Mostrará por la consola:

```
Domingo
Martes
```

## Caso 2: Pasar de cadena a enum

```
diaSemana dia;

if (Enum.TryParse("Viernes", true, out dia))
    Console.WriteLine(dia);

dia = (DiaSemana)Enum.Parse(enumType: typeof(DiaSemana), value: "lunes", ignoreCase: true);
Console.WriteLine(dia);

dia = (DiaSemana)Enum.Parse(enumType: typeof(DiaSemana), value: "Monday", ignoreCase: true); // ✗ ERRO
Console.WriteLine(dia);
```

Mostrará por la consola:

```
Viernes
Lunes
Unhandled exception. System.ArgumentException: Requested value 'Monday' was not found.
```

Fíjate que al convertir de cadena a enum, si la cadena no está en el enum se producirá un error, por lo que es recomendable usar `Enum.TryParse` para evitar excepciones.

## Caso 3: Pasar de enum a entero

```
DiaSemana dia = DiaSemana.Sábado;
int valorDia = (int)dia;
Console.WriteLine(valorDia);
```

Mostrará por la consola:

```
5
```

## Caso 4: Pasar de entero a enum

```
DiaSemana dia;
dia = (DiaSemana)5;
Console.WriteLine(dia);
```

Mostrará por la consola:

Sábado

# Métodos de utilidad para enumeraciones

- `static Array Enum.GetValues(Type enum)`  
Me devuelve un array del valor enumerado del tipo.
- `static string[] Enum.GetNames(Type enum)`  
Me devuelve un array de cadenas con los valores posibles del enum.
- `static bool Enum.IsDefined(Type enum, object value)`  
Me dice si value está en el enum en alguna de sus formas (enum, int, string).

## Ejemplo 1:

Supongamos que tenemos el siguiente enum con los días de la semana donde hemos asignado un valor explícito a cada día empezando por 1.

```
public class Ejemplo
{
    public enum DiaSemana
    {
        Lunes = 1, Martes = 2, Miércoles = 3, Jueves = 4, Viernes = 5, Sábado = 6, Domingo = 7
    }

    public static void Main()
    {
        DiaSemana[] diasSemana = (DiaSemana[])Enum.GetValues(typeof(DiaSemana));

        foreach (DiaSemana dia in diasSemana)
            Console.WriteLine($"{dia} = {(int)dia}");

        string[] nombresDiasSemana = Enum.GetNames(typeof(DiaSemana));
        Console.WriteLine(string.Join(", ", nombresDiasSemana));

        string diaBuscado = "Juernes";
        bool existe = Enum.IsDefined(typeof(DiaSemana), diaBuscado);
        Console.WriteLine(
            $"El día '{diaBuscado}' {(existe ? "sí" : "no")}" + existe ? " está definido en el enum DiaSemana." : "");
    }
}
```

Mostrará por la consola:

```
Lunes = 1
Martes = 2
Miércoles = 3
Jueves = 4
Viernes = 5
Sábado = 6
Domingo = 7
Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo
El día 'Juernes' no está definido en el enum DiaSemana.
```

## Ejemplo 2:

Vamos a implementar un método denominado **PresupuestoAnual**, que devuelva el presupuesto anual en euros, de los diferentes departamentos de una empresa ficticia.

Los posibles departamentos serán **Marketing, Compras, Ventas, RRHH, Administración** y su presupuesto será un valor literal de tu elección.

```
class Ejemplo
{
    public enum Departamento
    {
        Marketing, Compras, Ventas, RRHH, Administración
    }

    public static double PresupuestoAnual(in Departamento d) => d switch
    {
        Departamento.Marketing => 30000d,
        Departamento.Compras or Departamento.Ventas => 40000d,
        Departamento.RRHH => 10000d,
        Departamento.Administración => 25000d,

        // Si en el futuro añadimos un nuevo departamento a nuestra enumeración
        // nos avisará con un error.
        // Nota: El tratamiento de errores lo veremos más adelante.
        _ => throw new NotImplementedException("Falta por tener en cuenta un departamento")
    };
}
```

```

public static void Main()
{
    Departamento departamento;
    bool enumCorrecto;
    do
    {
        Console.WriteLine("Departamento: ");
        enumCorrecto = Enum.TryParse(Console.ReadLine(), true, out departamento);
        if (!enumCorrecto)
            Console.WriteLine("Prueba otra vez con " +
                $"{string.Join(", ", Enum.GetNames(typeof(Departamento)))}");
    } while (!enumCorrecto);

    Console.WriteLine("El presupuesto anual para " +
        $"{departamento.ToString().ToLower()} es de " +
        $"{PresupuestoAnual(departamento)} euros.");
}
}

```

Mostrará por la consola:

```

Departamento: Finanzas
Prueba otra vez con Marketing, Compras, Ventas, RRHH, Administración
Departamento: Compras
El presupuesto anual para compras es de 40000 euros.

```

## Ampliación opcional:

Aunque el tipo enumerado como entero de C# es equivalente al de C y C++. En otros lenguajes se implementa de otras formas de forma conceptual. Por ejemplo en **Kotlin** y **Python** son clases con un conjunto de constantes, y por tanto se comportan como objetos. Pero la palabra **enum** sigue apareciendo y solo tendremos que averiguar cómo se definen y usan. Veamos pues cómo definir el enum del ejemplo anterior y la función `PresupuestoAnual` en otros lenguajes como **Kotlin** y **Python** para que puedas **reconocer equivalencias** con **C#**.

### **Kotlin:**

```
enum class Departamento {
    Marketing, Compras, Ventas, RRHH, Administracion
}
fun presupuestoAnual(d: Departamento): Double = when (d) {
    Departamento.MARKETING -> 30000.0
    Departamento.COMPRAS, Departamento.VENTAS -> 40000.0
    Departamento.RRHH -> 10000.0
    Departamento.ADMINISTRACION -> 25000.0
}
```

### **Python:**

```
from enum import Enum, auto
class Departamento(Enum):
    Marketing = auto()
    Compras = auto()
    Ventas = auto()
    RRHH = auto()
    Administracion = auto()

def presupuesto_anual(d: Departamento) -> float:
    match d:
        case Departamento.Marketing:
            return 30000.0
        case Departamento.Compras | Departamento.Ventas:
            return 40000.0
        case Departamento.RRHH:
            return 10000.0
        case Departamento.Administracion:
            return 25000.0
        case _:
            raise NotImplementedError("Falta por tener en cuenta un departamento")
```

# Enumeraciones NO excluyentes (Máscara de Bits de estado o Flags)

## Enlaces

### Enumeraciones como bits de estado o flags

El concepto es muy similar al de flag que vimos con booleanos. Por tanto, es una forma **compacta** y **muy rápida** de guardar varios **flag de estado** asociándolos a un **bit** en memoria en lugar de a una variable booleana.

Por ejemplo, el valor binario de un byte en memoria puede ser `01100111` y cada bit puede ser un 'flag' con un significado donde el `1` significa que se cumple y el `0` que no.

Además, **para nombrar o identificar** el significado de **los 'flags' asociados a un bit** utilizaremos una **enumeración**.

Veamos la sintaxis y uso a través de un ejemplo...

Supongamos la siguiente enumeración **no excluyente** para gestionar los extras en cierto modelo de coche...

```
[Flags]
enum Extra : byte
{
    None          = 0b_0000_0000,    // 0
    Climatizador = 0b_0000_0001,    // 1
    Navegador     = 0b_0000_0010,    // 2
    FullLed       = 0b_0000_0100,    // 4
    LlantasDeportivas = 0b_0000_1000, // 8
}
```

Fíjate que hemos añadidos el **atributo** o '**anotación**' `[Flags]` sobre la definición de la enumeración para indicar que vamos a definir los nombres de los flags.

Además, hemos hecho que el entero subyacente sea de tipo `byte` y **hemos definido por extensión con un literal binario** (`0b_`), los valores de cada byte a las **potencias de 2** de tal manera que realizará la asociación entre el valor enumerado y el 'flag' que representa en la byte.

En un principio la byte estará todo a ceros, a través de la asignación y para cambiarlo utilizaremos **operaciones de bit**.

```
Extra extras = default; // default equivale a Extra.None
```

## 1. Si queremos añadir uno o varios extras al coche usaremos el **or de bit** |

```
extras |= Extra.Climatizador | Extra.FullLed;  
Console.WriteLine(extras);  
Console.WriteLine("{Convert.ToString((byte)extras, 2).PadLeft(8, '0')}");
```

Estaremos haciendo la operación:

```
00000000 (None)  
00000001 (Climatizador)  
OR 00000100 (FullLed)  
  
00000101 (Climatizador | FullLed)
```

Mostrará por la consola:

```
climatizador, FullLed  
00000101
```

## 2. Si queremos ver si hemos establecido algún extra al coche usaremos el **and de bit** & Fíjate que al usar enumerados la operación es mucho más legible.

```
// Hay que tener cuidado con la prioridad de & y por eso ponemos paréntesis.  
bool tieneClimatizador = (extras & Extra.Climatizador) == Extra.Climatizador;  
Console.WriteLine(tieneClimatizador);
```

Estaremos haciendo la operación:

```
00000101 (Climatizador | FullLed)  
AND 00000001 (Climatizador)  
  
00000001 (Climatizador)
```

Mostrará por la consola:

```
True
```

## 3. Si queremos quitar algún extra al coche usaremos el **and de bit** & y la **negación de bit** ~

```
extras &= ~Extra.Climatizador;
Console.WriteLine(extras);
Console.WriteLine("{Convert.ToString((byte)extras, 2).PadLeft(8, '0')}");
```

Estaremos haciendo la operación:

```
00000101 (Climatizador | FullLed)
AND 11111110 (~Climatizador)

_____
00000100 (FullLed)
```

Mostrará por la consola:

```
FullLed
00000100
```

### Ejemplo:

En el siguiente código vamos a definir una enumeración no excluyente para almacenar los estados combinados de un juego de plataformas. De tal manera que **la primera letra** del 'flag' me va a **activar o desactivar** dicho estado (*El enumerado del estado debería empezar por una letra diferente*). Mostrándome tras cada pulsación como se encuentran los flags, tanto el valor de enumerado como el valor interno en binario del enum.

Además, indicaremos que teclas activarán o desactivarán un determinado estado.

 **Nota:** Fíjate como el código se ha implementado para que funcione, independientemente del nombre que hemos asignado al flag en la enumeración y del número de flags que tengamos definidos.

```
public class Ejemplo
{
    [Flags]
    public enum PlayerState : byte
    {
        None      = 0b_0000_0000,    // 0
        PowerUp   = 0b_0000_0001,    // 1
        Walking   = 0b_0000_0010,    // 2
        Jumping   = 0b_0000_0100,    // 4
        Attacking = 0b_0000_1000,    // 8
        Shield    = 0b_0001_0000,    // 16
    }
}
```

```

public static string GameOptions()
{
    string options = "Game keys ( ";
    foreach (PlayerState playerState in Enum.GetValues(typeof(PlayerState)))
        options += $"'{playerState.ToString()[0]}'} = {playerState} ";
    options += ") Press E to Exit.";
    return options;
}

public static PlayerState StateAccordingToKey(char key)
{
    PlayerState stateForKey = PlayerState.None;
    foreach (PlayerState s in (PlayerState[])Enum.GetValues(typeof(PlayerState)))
    {
        if (s.ToString()[0] == key)
        {
            stateForKey = s;
            break;
        }
    }
    return stateForKey;
}

public static void Main()
{
    Console.CursorVisible = false;
    char key;
    PlayerState state = default;
    string gameOptions = GameOptions();
    do
    {
        Console.WriteLine($"PlayerStarte = {state} ({Convert.ToString((byte)state, 2).PadLeft(8, '0')}");
        Console.WriteLine(gameOptions);
        key = char.ToUpper(Console.ReadKey(true).KeyChar);
        PlayerState stateToSwitch = StateAccordingToKey(key);
        if ((state & stateToSwitch) == stateToSwitch)
            state &= ~stateToSwitch;
        else
            state |= stateToSwitch;
    }
    while (key != 'E');
}
}

```