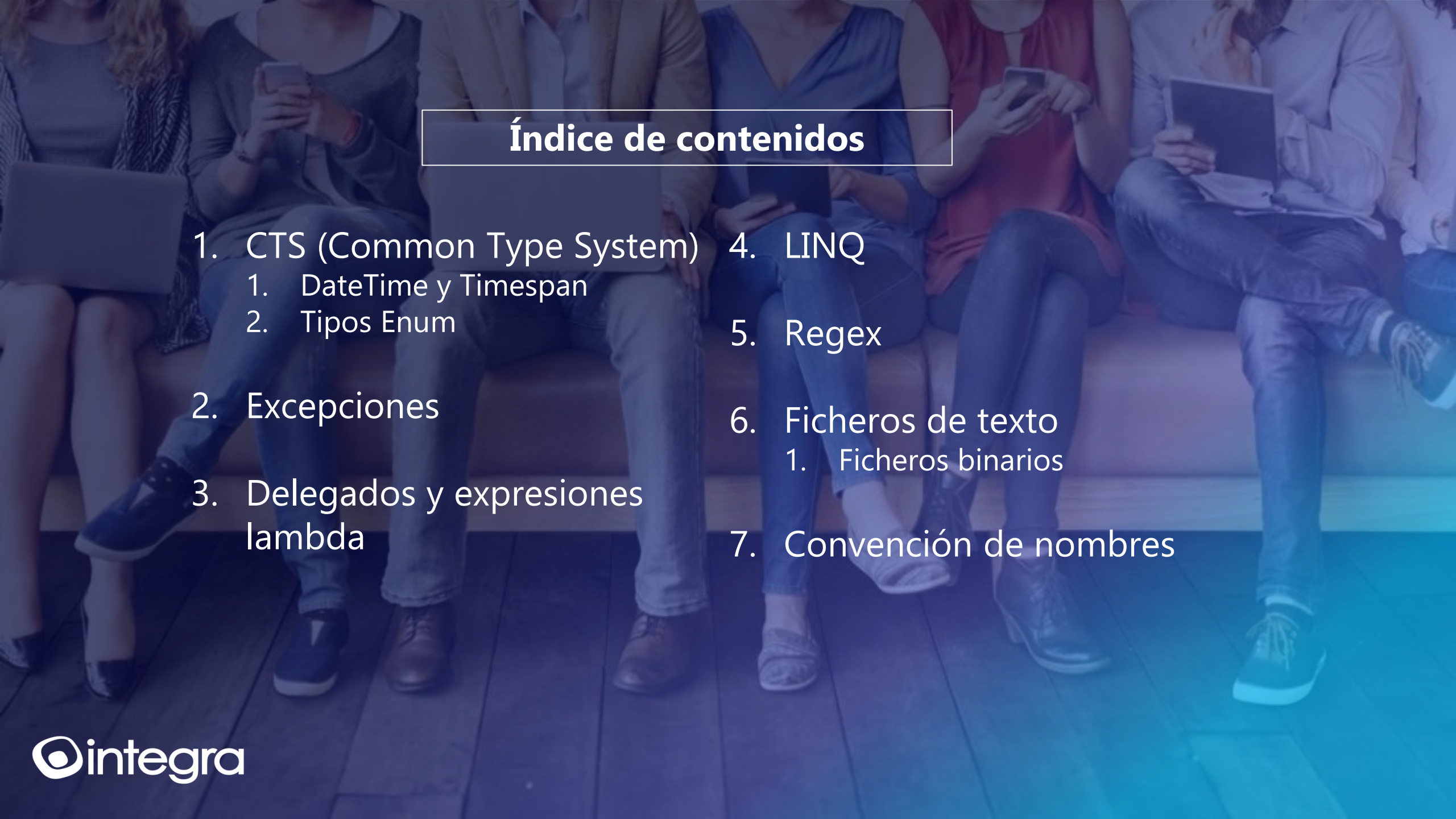
Módulo 2

El lenguaje C# avanzado

Tecnara – Cursos de formación



Índice de contenidos

integra


tecnara
Cluster TIC de Aragón / Aragón IT Cluster
01
CTS
integra
Programación web con .NET | El lenguaje C# avanzado

CTS (Common Type System)

- C# tiene unos tipos primitivos predefinidos (int, string, etc…)
- .NET Framework define muchas clases que podemos usar como tipos en nuestros programas
- Además, nosotros podemos crear nuevas clases y hacer que hereden de clases ya existentes si lo necesitamos
- Y todo esto… ¿Cómo funciona?

CTS (Common Type System)

- 1º punto: herencia. Existe la herencia entre clases, e igual que hay clases existentes que heredan de otras, también admiten herencia las nuevas clases creadas por nosotros

```
public sealed class DisplayAttribute : Attribute

public class Tecnico : TrabajadorDepTecnologia
```

CTS (Common Type System)

- 1º punto: herencia. Existe la herencia entre clases, e igual que hay clases existentes que heredan de otras, también admiten herencia las nuevas clases creadas por nosotros

- 2º punto: Cada tipo en C# es un tipo pasado por valor o un tipo pasado por referencia

  - Tipo pasado por valor: al asignar una variable a otra, se copia el VALOR de la variable. Si se modifica el valor de la variable original, la variable asignada **no cambia** también su valor. Un tipo pasado por valor NO SE PUEDE HEREDAR

```
int num1 = 5;
int num2 = num1;
num1 = 10;
Console.WriteLine(num2); // Escribirá por pantalla el nº 5
```

CTS (Common Type System)

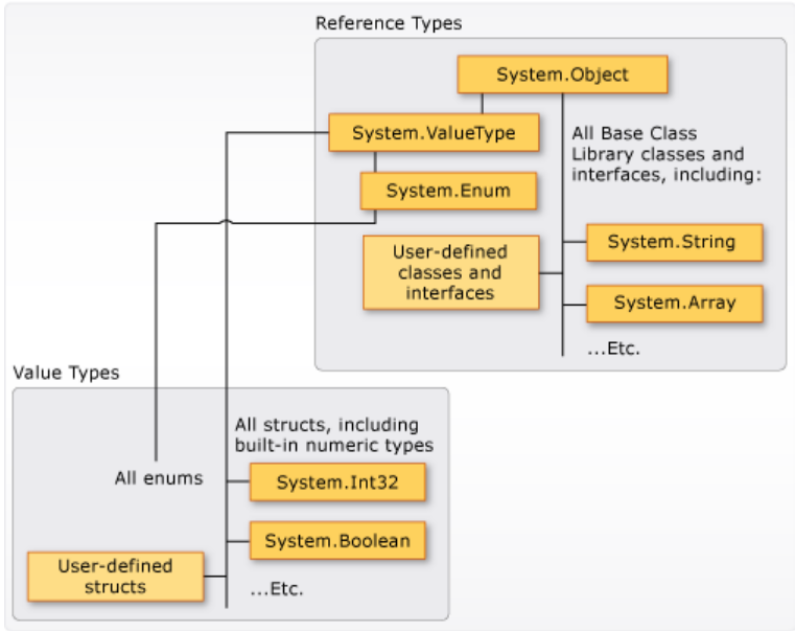
- 1º punto: herencia. Existe la herencia entre clases, e igual que hay clases existentes que heredan de otras, también admiten herencia las nuevas clases creadas por nosotros

- 2º punto: Cada tipo en C# es un tipo pasado por valor o un tipo pasado por referencia

  - Tipo pasado por referencia: al asignar una variable a otra, se copia la REFERENCIA de la variable. Si se modifica el valor de la variable original, la variable asignada **sí cambia** también su valor. Un tipo pasado por referencia SÍ SE PUEDE HEREDAR

CTS (Common Type System)

- 2º punto: Cada tipo en C# es un tipo pasado por valor o un tipo pasado por referencia:
  - Tipos pasados por valor: tipos struct (int, decimal, bool, char, …) y tipos enum
  - Tipos pasados por referencia: tipos class (string, List, clases nuevas…), interfaces y delegados



CTS (Common Type System)

- Tipos predefinidos en C#: 2 por referencia, 13 por valor

C# type keyword	.NET type
object	System.Object
string	System.String

C# type keyword	.NET type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
short	System.Int16
ushort	System.UInt16

CTS (Common Type System)

- Elementos especiales del CTS:
  - Los literales también tienen tipo, y se pueden aplicar métodos del tipo sobre el literal
    string s = $“el tipo del número 5 es {5.GetType().Name()}”;

  - Tipos genéricos: se puede crear una clase como List, que está definida como List<T>, donde T puede ser cualquier tipo, por valor o por referencia. Pero hay que saber como usarla…

  - Tipos implícitos: var x = 5; var y = “hola”;
  - tipos anónimos: var z = new { Edad = 35, Nombre = “Paco” };
  - tipos nulables (nullable types): int? enteroNulable1 = 5; int? enteroNulable2 = null;
    - operador ‘??’ para asignar un valor no nulo si es nulo: int entero1 = enteroNulable2 ?? -1;

DateTime y Timespan

- Fechas y horas: Asignación de expresiones (primero se hacen las operaciones a la derecha del símbolo '=', y el resultado se asigna a la variable a la izquierda del '=')
- DateTime date = DateTime.Now; // aquí date tendrá la fecha/hora actual
- DateTime date = DateTime.MinValue // día 01/01/0001 00:00:00
- string strDate = DateTime.Now.ToString("dd/MM/yyyy");
  // string con la fecha actual, mostrando el día, mes y año separados por '/'
- Más formatos para ToString() en https://www.c-sharpcorner.com/blogs/date-and-time-format-in-c-sharp-programming1 (mientras exista la página)

DateTime y Timespan

- Otros operadores para fechas y horas
- DateTime fecha = new DateTime(2019, 9, 17, 10, 35, 41)
- fecha.Year // devuelve el año 2019 como variable de tipo int
  (igual que Year, existen: Month, Day, Hour, Minute, Second, Millisecond)
- fecha.AddDays(7) // devuelve la fecha con 7 días más
- fecha.AddDays(-7) // devuelve la fecha con 7 días menos
  (Más: AddYears, AddMonths, AddHours, AddMinutes, AddSeconds, AddMilliseconds)

DateTime y Timespan

- Otros operadores para fechas y horas
- DateTime fecha = new DateTime(2019, 9, 17, 10, 35, 41)
- fecha.DayOfWeek // devuelve un tipo enum con los días de la semana (DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday, DayOfWeek.Thursday, DayOfWeek.Friday, DayOfWeek.Saturday, DayOfWeek.Sunday)

DateTime y Timespan

Otros operadores para fechas y horas: clase Timespan

```
Timespan ts = new Timespan(2, 59, 45) // período de 2h 59min 45s
Timespan ts = new Timespan(3, 2, 59, 45) // período de 3 días 2h 59min 45s

DateTime fecha = new DateTime(2019, 9, 17, 10, 35, 41)

DateTime d1 = DateTime.Now;
DateTime d2 = DateTime.Now.AddDays(7);
// el uso principal de Timestamp: tiempo pasado entre dos fechas/horas
Timespan diff = d2-d1;
Console.WriteLine(diff);
```

Tipos enum

- Un tipo enumerado es un tipo por valor definido por un conjunto de constantes con nombre textual a las que se les asigna un valor numérico que puede ser implícito si da igual, o puede ser explícito si es importante. El valor por defecto del primer elemento es 0, del segundo 1, etc…

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

- Por defecto los valores numéricos asociados son de tipo int, pero se puede cambiar ese tipo usando la sintaxis para la herencia de clases

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

Tipos enum

- Declaración de tipos enumerados
- Se accede como: Color.Green, Color.Blue
- Ejemplos ya vistos:
  enum ConsoleColor: ConsoleColor.Yellow
  (para cambiar color de texto en consola)
- enum DayOfWeek: DayofWeek.Monday
  (para identificar el día lunes de la semana)

```
public enum Color
{
    Green,   //defaults to 0
    Orange,  //defaults to 1
    Red,     //defaults to 2
    Blue     //defaults to 3
}

enum WeekDays
{
    Monday = 0,
    Tuesday =1,
    Wednesday = 2,
    Thursday = 3,
    Friday = 4,
    Saturday =5,
    Sunday = 6
}

Console.WriteLine(WeekDays.Friday);
Console.WriteLine((int)WeekDays.Friday);
```

Tipos enum

- Uso de los enums como máscaras de bits: asignar valores que sean potencias de dos. Esto permite usar varias constantes de tipo enum a la vez para crear una máscara con OR y AND

```
[Flags]
public enum Days
{
    None      = 0b_0000_0000,  // 0
    Monday    = 0b_0000_0001,  // 1
    Tuesday   = 0b_0000_0010,  // 2
    Wednesday = 0b_0000_0100,  // 4
    Thursday  = 0b_0000_1000,  // 8
    Friday    = 0b_0001_0000,  // 16
    Saturday  = 0b_0010_0000,  // 32
    Sunday    = 0b_0100_0000,  // 64
    Weekend   = Saturday | Sunday
}
```

```
public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays & workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) == Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday: {isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}
```

Tipos enum

- Conversión de una constante con nombre de tipo enum en su valor numérico asociado: se obliga a un casting explícito

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output: Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

02

Excepciones

Excepciones

- En el tema 1 del módulo 1 se dijo que cuando el procesador ejecuta una instrucción que genera un error, el resultado devuelto es una excepción, que si no se controla cierra el programa
- Esa excepción se puede controlar para que no se cierre el programa
- Para hacerlo bien, se debe ejecutar algún tipo de código que 'solucione el error' o comunique el error de alguna manera antes de continuar con el programa

Excepciones

- Existen muchísimas excepciones ya definidas en .NET Framework
- Además, se pueden crear excepciones personalizadas si así se desea
- En la documentación de los métodos de .NET Framework aparecen las excepciones que se pueden generar durante su ejecución
- Si no se conocen las excepciones concretas que se pueden generar, todas heredan de Exception, así que si se controla Exception, se controlan todas
- Aunque controlar excepciones específicas es más útil en casos concretos

Excepciones

- Sintaxis de un control de excepción:

```
try {
      // Código que puede generar una excepción
} catch (Exception e) { // se puede cambiar Exception por una excepción específica
      // código que controla la excepción
} finally { // el bloque finally es opcional
      // aquí aparece código que se ejecutará tanto si hay excepción como si todo va bien
}
```

Excepciones

- Hasta ahora estábamos controlando los posibles errores comprobando con ifs que el valor de las variables no fuese a generar error.

- Ahora podemos hacer eso de otra forma, escribiendo solo el código que supone que todo va bien, metido en un bloque try-catch que controle el caso de error

Excepciones

- Ejemplo sin control de excepciones:

```
for (int intentos = 1; intentos <= 3; intentos++)
{
    // Obtener del usuario los valores por pantalla
    Console.Write("Introduce la cantidad que desea ingresar: ");
    string srt_ingreso = Console.ReadLine();

    // Intentar transformar los strings leídos en valores decimales
    decimal ingreso;
    bool str_ingresoEsDecimal = decimal.TryParse(srt_ingreso, out ingreso);

    if (str_ingresoEsDecimal == false)
    {
        // si el usuario introduce un valor erróneo, se lo indicamos en pantalla, con texto en color rojo
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Error en el valor introducido");
        Console.ResetColor();
    }
    else
    {
        // si el usuario introduce un valor correcto como decimal, ingresar esa cantidad a la cuenta
        saldo += ingreso;
        listaIngresos.Add(ingreso);
        listaMovimientos.Add(ingreso);
        Console.WriteLine($"La operación se ha realizado correctamente, su nuevo saldo es {saldo}€");
        break;
    }
}
```

Excepciones

- Ejemplo con control de excepciones:

```
for (int intentos = 1; intentos <= 3; intentos++)
{
    // Obtener del usuario los valores por pantalla
    Console.Write("Introduce la cantidad que desea ingresar: ");
    try
    {
        decimal ingreso = decimal.Parse(Console.ReadLine());

        saldo += ingreso;
        listaIngresos.Add(ingreso);
        listaMovimientos.Add(ingreso);
        Console.WriteLine($"La operación se ha realizado correctamente, su nuevo saldo es {saldo}€");
        break;
    }
    catch (Exception)
    {
        // si el usuario introduce un valor erróneo, se lo indicamos en pantalla, con texto en color rojo
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine("Error en el valor introducido");
        Console.ResetColor();
    }
}
```

Excepciones

- Si se quiere forzar a que se genere una excepción, existe la instrucción throw (lanzar):

throw Exception;

- Si se lanza una excepción en un método que no la controla con un bloque try-catch, el método termina y la excepción se lanza para el método invocante, con el mismo comportamiento: si no la controla, acaba el método y pasa al método invocante.

- Si no la controla ningún método hasta el Main de Program, se acaba el programa

Excepciones

- La clase Exception tiene un atributo Message que habitualmente contiene información textual que describe la razón que ha causado la excepción. Esta información se puede usar para saber como programadores qué ha producido el error y facilitar su corrección, o en programas que empleen un sistema de log de errores, se puede guardar ese error en el sistema (normalmente con la fecha/hora, el usuario si existe, y a veces hasta con el valor de las variables involucradas en la generación del error)

- En algunos casos de excepciones muy generales, la excepción tiene otra excepción en su interior, en su atributo InnerException. En esos casos, la información detallada aparece en el atributo Message de su InnerException

tecnara
Cluster TIC de Aragón / Aragón IT Cluster

03

Delegados y expresiones lambda

integra

Delegados y expresiones lambda

- Además de struct, enum, class e interface, existe un quinto tipo de declaración de tipo que se puede realizar dentro de un namespace o dentro de una clase: **delegate**

- Un tipo delegado es como una interfaz pensada solo para un método.

```
public delegate void Del(string message);
```

- Si se pueden definir variables de un tipo que sea interfaz, pero su valor asignado debe ser de alguna clase que implemente esa interfaz, para los delegados ocurre algo parecido: se pueden definir variables de un tipo que sea delegado, pero su valor asignado debe ser algún método que cumpla la tipología definida por el delegado (mismo nº y tipo de parámetros de entrada, mismo tipo de resultado devuelto)

Delegados y expresiones lambda

- Se pueden definir variables de un tipo que sea delegado, pero su valor asignado debe ser algún método que cumpla la signatura definida por el delegado

```
public delegate void Del(string message);
```

```
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Delegados y expresiones lambda

- Como el valor guardado en la variable delegado es un objeto, se puede pasar como parámetro a otros métodos

```
public delegate void Del(string message);

public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}

// Instantiate the delegate.
Del handler = DelegateMethod;

public static void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}

MethodWithCallback(1, 2, handler);
```

Delegados y expresiones lambda

- No es necesario que el método implementado sea estático.

```
public delegate void Del(string message);

public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}

var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;
```

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Delegados y expresiones lambda

- ¡Se pueden añadir varios métodos a un único delegado, para que se ejecuten uno detrás de otro! Para ello existe el operador '+' para añadir un método a un delegado

```
public delegate void Del(string message);

public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}

var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Delegados y expresiones lambda

- Y también existe el operador '-' para quitar un método a un delegado que ya tiene varios incluidos

```
public delegate void Del(string message);

public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Delegados y expresiones lambda

- Si se quieren conocer los métodos incluidos en un delegado: GetInvocationList()
- Para conocer el nº de métodos del delegado: GetInvocationList().GetLength(0)

```
public delegate void Del(string message);

public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Delegados y expresiones lambda

- El uso de delegados es la base del diseño orientado a eventos, que es el empleado en el desarrollo de aplicaciones de Escritorio y desarrollo de APPs para móvil

- En aplicaciones de consola los eventos no tienen tanto sentido como en escritorio y móvil, porque la mayoría de los eventos de ambos tipos de desarrollo se producen por una interacción asíncrona del usuario con la interfaz del programa (el usuario hace click en un botón, o hace scroll hacia abajo de la pantalla, o mueve el ratón dentro de un área determinada de la pantalla…). Todas estas acciones no se pueden realizar en consola, porque no hay click, no hay scroll y no hay movimiento del puntero. En consola el único evento que se puede capturar es la escritura por teclado con Read, ReadLine y ReadKey

Delegados y expresiones lambda

- En el ejemplo anterior hemos creado un delegado Del(string message). También se pueden crear delegados cuyos parámetros no sean de un tipo específico, sino de un tipo genérico (como ocurría con List<T>, pero aplicado a delegados)

- C# define varios de estos delegados con parámetros genéricos, y los separa en dos grupos: los que no devuelven valor (Action) y los que devuelven un único valor (Func)
  - Para cada grupo define 17 delegados, con entre 0 y 16 parámetros de entrada
  - Muchos métodos de C# definen delegados de tipo Func como parámetros
  - Entre los métodos que definen delegados como parámetros están los métodos de LINQ

Delegados y expresiones lambda

Func<TResult>

Func<T,TResult>

Func<T1,T2,TResult>

Func<T1,T2,T3,TResult>

Func<T1,T2,T3,T4,TResult>

Func<T1,T2,T3,T4,T5,TResult>

Func<T1,T2,T3,T4,T5,T6,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,TResult>

Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>

Action

Action<T>

Action<T1,T2>

Action<T1,T2,T3>

Action<T1,T2,T3,T4>

Action<T1,T2,T3,T4,T5>

Action<T1,T2,T3,T4,T5,T6>

Action<T1,T2,T3,T4,T5,T6,T7>

Action<T1,T2,T3,T4,T5,T6,T7,T8>

Action<T1,T2,T3,T4,T5,T6,T7,T8,T9>

Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10>

Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11>

Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12>

Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13>

Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14>

Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15>

Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>

Delegados y expresiones lambda

- En los ejemplos anteriores siempre hemos asignado al delegado métodos definidos (estáticos o no) que tenían un nombre, pero también se pueden asignar a tipos delegados lo que se conoce como funciones anónimas (Anonymous functions)

- Las funciones anónimas se pueden declarar de dos formas:
  - Mediante método anónimo, también llamado operador delegate (no recomendado)
  - Mediante expresión lambda, que hace lo mismo que el anterior, con una sintaxis más corta

Delegados y expresiones lambda

- Una expresión lambda es una forma corta de crear un método que no tenga nombre: sólo parámetros de entrada y código que genere el resultado

- Las funciones lambda están pensadas para códigos que generen el resultado en una o pocas líneas. Existen dos sintaxis diferentes para una expresión lambda:
  - Para una línea → (input-parameters) => expression
  - Para varias líneas → (input-parameters) => { <sequence-of-statements> }

Delegados y expresiones lambda

Los parámetros de entrada (parte izquierda de la flecha) siguen estas reglas:

- Si no hay parámetros, apertura y cierre de paréntesis: ()

```
Action line = () => Console.WriteLine();
```

- Si hay un parámetro, un nombre que lo identifique en la parte derecha de la flecha, opcionalmente el tipo de ese parámetro (normalmente los tipos de cada parámetro son inferidos por C# durante la ejecución), y opcionalmente paréntesis (el caso de un parámetro es el único en el que los paréntesis son opcionales)

```
Func<int, int> square = x => x * x;
```

- Si hay más de un parámetro, nombres que los identifiquen en la parte derecha de la flecha, separados por comas, opcionalmente los tipos de los parámetros, y todo entre paréntesis

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Delegados y expresiones lambda

El cuerpo de la expresión sigue estas reglas:

- Si solo tiene una línea
  - La expresión puede ir o no entre llaves, opcionalmente
  - El resultado de la expresión debe ser del tipo indicado en el delegado al que se va a asignar

```
Func<int, int> square = x => x * x;
```

- Si tiene más de una línea
  - La expresión debe ir entre llaves obligatoriamente
  - El resultado de la expresión debe ser del tipo indicado en el delegado al que se va a asignar

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

integra

Delegados y expresiones lambda

- Muchos de los métodos de LINQ necesitan un parámetro que sea de tipo Func<TSource, bool>. Esto significa que para ese parámetro hay que pasarle al método una expresión lambda que devuelva un resultado booleano para cada elemento de la enumeración sobre la que se vaya a aplicar ese método de LINQ.

- El parámetro TSource usado como parámetro de entrada a la expresión lambda es un tipo genérico, que al momento de ejecutarse el método, pasa a ser del tipo de los elementos de la enumeración que se esté recorriendo. Así:
  - para un Where(x => x > 18) aplicado a una lista de enteros, x se entenderá como de tipo int
  - para un Where(x => x.Edad > 18) aplicado a una lista de objetos Persona, x se entenderá como de clase Persona

04

LINQ

LINQ

- LINQ (Language INtegrated Query) es, como su nombre en inglés indica, un sub-lenguaje incluido en C# que permite realizar búsquedas y otras operaciones sobre colecciones de datos (estructuras, como List o Array, que almacenan un grupo de variables del mismo tipo)

- Este sub-lenguaje tiene dos variantes:
  - query syntax (sintaxis de consulta): muy similar al lenguaje SQL de consultas a Base de Datos
  - method syntax (sintaxis de método): invocación a métodos con expresiones lambda como parámetro

- En el curso veremos la sintaxis de método porque la otra tiene alguna diferencia con SQL, y si vamos a ver SQL más adelante no conviene conocer las dos para evitar líos

LINQ

- El tipo de variable sobre el que se pueden ejecutar los métodos de LINQ es Ienumerable
- Como List hereda de IEnumerable, también se pueden ejecutar sobre ella métodos LINQ
- Para tener disponibles los métodos propios de LINQ desde cualquier objeto de clase List<T>, es necesario incluir la sentencia using del namespace de la biblioteca de LINQ:

  using System.Linq;

LINQ

- Tipos de operaciones que permite LINQ sobre variables IEnumerable o clases hijas:

  - Búsqueda de un elemento
  - Filtrado de varios elementos
  - Ordenación de elementos
  - Operaciones de proyección
  - Partir en trozos la enumeración
  - Operaciones de conjuntos
  - Cuantificadores de elementos
  - Operadores JOIN
  - Operadores de agrupación
  - Generación de enumeraciones
  - Operador de igualdad entre enumeraciones
  - Operadores de conversión de tipo
  - Operador de concatenación
  - Operadores de agregación

LINQ

- Operadores de búsqueda de un elemento
  - **ElementAt(int)**: elemento que tiene un determinado índice en la enumeración
  - ElementAtOrDefault(int): mismo que anterior, o valor por defecto si el índice está fuera de rango
  - First(): primer elemento de la enumeración
  - First(Func<TSource, bool>): primer elemento de la enumeración que cumple una condición
  - FirstOrDefault(): primer elemento, o un valor por defecto si no lo encuentra
  - **FirstOrDefault(Func<TSource, bool>)**: primer elemento que cumpla condición, o valor por defecto
  - Last(): último elemento de la enumeración
  - Last(Func<TSource, bool>): último elemento de la enumeración que cumple una condición
  - LastOrDefault(): último elemento, o un valor por defecto si no lo encuentra
  - **LastOrDefault(Func<TSource, bool>)**: último elemento que cumpla condición, o valor por defecto
  - Single(): único valor de la lista, o error si hay cero o más de uno
  - Single(Func<TSource, bool>): único valor que cumpla condición, o error si hay cero o más de uno que cumplen
  - SingleOrDefault(): único valor de la lista, valor por defecto si no hay, error si hay más de uno
  - SingleOrDefault(Func<TSource, bool>): lo mismo que el anterior, pero para un valor que cumpla condición

LINQ

- Operadores de filtrado de varios elementos
  - OfType(TResult): elementos que son del tipo especificado
  - **Where(Func<TSource, bool>)**: elementos que cumplen condición
  - Where(Func<TSource, int, bool>): lo mismo que el anterior, pero se incluye el índice entre las variables

LINQ

- Operadores de ordenación de elementos
  - **OrderBy(Func<TSource, TKey>)**: ordenación ascendente por la clave seleccionada
  - OrderBy(Func<TSource, TKey>, IComparer<TKey>): igual, pero la ordenación está definida por el comparador
  - **OrderByDescending(Func<TSource, TKey>)**: ordenación descendente por la clave seleccionada
  - OrderByDescending (Func<TSource, TKey>, IComparer<TKey>): igual, con ordenación según comparador
  - ThenBy(Func<TSource, TKey>): ordenación extra ascendente por la clave seleccionada
  - ThenBy(Func<TSource, TKey>, IComparer<TKey>): igual, con ordenación según comparador
  - ThenByDescending(Func<TSource, TKey>): ordenación extra descendente por la clave seleccionada
  - ThenByDescending(Func<TSource, TKey>, IComparer<TKey>): igual, con ordenación según comparador
  - **Reverse()**: elementos actuales en orden inverso

LINQ

- Operadores de proyección de elementos

  El concepto de proyección se refiere a partir de una lista de elementos pertenecientes a una clase con varios atributos, y a partir de ella obtener otra lista que contenga el mismo nº de elementos que la lista original, donde cada elemento, en vez de contener el elemento original, contenga:
  - uno de sus atributos, o
  - varios de sus atributos, o
  - una operación hecha a partir de uno o varios de sus atributos, o
  - varias operaciones hechas a partir de uno o varios de sus atributos

LINQ

- Ejemplo. Partimos de
- ListaPersonas={{name="Pepe", surname="Pérez", Edad=37},
  {name="Ana", surname="Buil", Edad=24}}

Proyección de:

- uno de sus atributos: ListaNombres={{"Pepe"}, {"Ana"}}
- varios de sus atributos: ListaNombresEdad={{"Pepe", 37}, {"Ana", 24}}
- operación a partir de un atributo: ListaANombres={{"APepe"}, {"AAna"}}
- operación a partir de varios atributos: ListaNS={{"Pepe Pérez"}, {"Ana Buil"}}
- operaciones a partir de un atributo: ListaDigitosEdades={{3, 7}, {2, 4}}
- operaciones a partir de varios atributos: ListRara={{"Pep3","Pér7"}, {"Ana2","Bui4"}}

LINQ

- Operadores de proyección de elementos
  - **Select(Func<TSource, TResult>)**: lista que contiene una proyección de cada elemento de la lista original
  - Select(Func<TSource, int, TResult>): lo mismo, pero el índice tiene utilidad en la proyección
  - SelectMany(Func<TSource, IEnumerable<TResult>>): cuando la proyección es sobre atributo tipo lista
  - SelectMany(Func<TSource, int, IEnumerable<TResult>>): igual, pero el índice tiene utilidad en la proyección
  - SelectMany(Func<TSource, IEnumerable<TCollection>>, Func<TSource, TCollection, TResult>): proyección sobre atributo tipo lista, y aún se desea hacer otra proyección sobre esa proyección intermedia
  - SelectMany(Func<TSource, int, IEnumerable<TCollection>>, Func<TSource, TCollection, TResult>): lo mismo, pero el índice tiene utilidad en la proyección intermedia

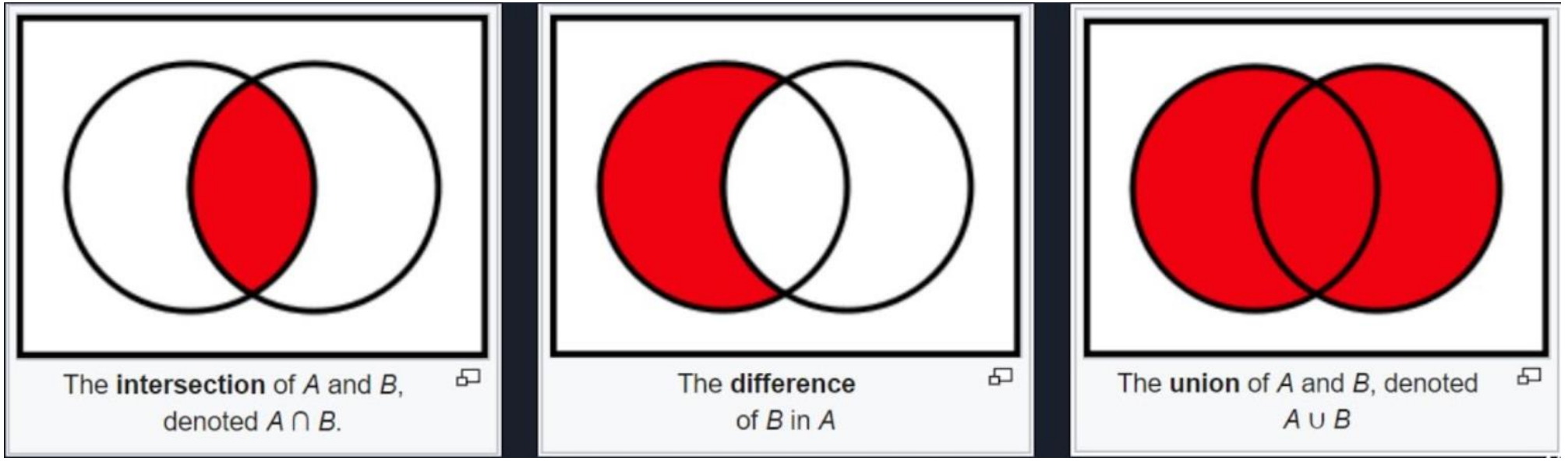
LINQ

- Operadores de partición de la enumeración
  - **Skip(int)**: lista sin los primeros n elementos (n lo indica el parámetro)
  - SkipWhile(Func<TSource, bool>): lista sin elementos del principio hasta que se deje de cumplir una condición
  - SkipWhile(Func<TSource, int, bool>): lo mismo, pero el índice tiene utilidad en la condición
  - **Take(int)**: lista solo con los primeros n elementos (n lo indica el parámetro)
  - TakeWhile(Func<TSource, bool>): lista de elementos del principio hasta que se deje de cumplir una condición
  - TakeWhile(Func<TSource, int, bool>): lo mismo, pero el índice tiene utilidad en la condición

LINQ

- Operadores de conjuntos

  Las operaciones binarias clásicas sobre conjuntos son: diferencia, unión e intersección



  C# incluye también la operación unaria de eliminación de elementos repetidos

LINQ

- Operadores de conjuntos
  - **Distinct()**: eliminación de elementos repetidos, usando el comparador de igualdad por defecto
  - Distinct(IEqualityComparer<TSource>): lo mismo, usando un comparador de igualdad personalizado
  - Except(IEnumerable<TSource>): quita de la enumeración inicial los de la enumeración pasada por parámetro
  - Except(IEnumerable<TSource>, IEqualityComparer<TSource>): igual, con comparador personalizado
  - Intersect(IEnumerable<TSource>): intersección entre inicial y pasada por parámetro
  - Intersect(IEnumerable<TSource>, IEqualityComparer<TSource>): igual, con comparador personalizado
  - Union(IEnumerable<TSource>): unión entre inicial y pasada por parámetro
  - Union(IEnumerable<TSource>, IEqualityComparer<TSource>): igual, con comparador personalizado

LINQ

- Operadores de cuantificación
  - All(Func<TSource, bool>): true/false según todos los elementos cumplen condición o alguno no
  - Any(): true/false según la enumeración tenga o no elementos
  - **Any(Func<TSource, bool>)**: true/false según la enumeración tenga o no algún elemento que cumpla condición
  - **Contains(TSource)**: true/false según si enumeración contiene o no elemento pasado por parámetro
  - Contains(TSource, IEqualityComparer<TSource>): lo mismo, usando un comparador de igualdad personalizado

LINQ

- Operadores Join. No se van a explicar aquí porque tienen que ver con Bases de datos
  - Join(IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)
  - Join(IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>, IEqualityComparer<TKey>)
  - GroupJoin(IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)
  - GroupJoin(IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>, IEqualityComparer<TKey>)

LINQ

- Operadores de agrupación. También tienen que ver con Bases de datos
  - GroupBy() con 8 sobrecargas
  - ToLookUp() con 4 sobrecargas

LINQ

- Operadores de generación de enumeraciones desde cero
  - DefaultIfEmpty() con 2 sobrecargas
  - Empty()
  - Range(int int)
  - Repeat(Tresult, int)

LINQ

- Operador de comprobación de igualdad entre enumeraciones
  - SequenceEqual() con 2 sobrecargas

LINQ

- Operadores de conversión de tipos
  - AsEnumerable()
  - AsQueryable() con 2 sobrecargas
  - ToArray()
  - ToDictionary() con 4 sobrecargas
  - ToList()
  - ToLookup() con 4 sobrecargas (ya visto en los operadores de agrupación)
  - Cast<TResult>()

LINQ

- Operador de concatenación de enumeraciones del mismo tipo
  - Concat(IEnumerable<TSource>)

LINQ

- Operadores de agregación
  - Aggregate() con 3 sobrecargas
  - Average() con 20 sobrecargas (para distintos tipos numéricos, nulables y no nulables)
  - Count() con 2 sobrecargas
  - LongCount() con 2 sobrecargas
  - Max() con 22 sobrecargas
  - Min() con 22 sobrecargas
  - Sum() con 20 sobrecargas

05

Regex

Expresiones regulares

- Una expresión regular (ER) es un patrón textual, es decir, una estructura léxico-gramatical que comparten varios textos o palabras posibles entre sí. Ejemplos:

- ER para un número: secuencia de uno o más caracteres entre el 0 y el 9
- ER para palabras que empiezan por a: un carácter ‘a’, seguido de cero o más letras (a-z)
- ER para dos palabras separadas por espacio en blanco: una o más letras (a-z), seguido de un espacio en blanco, seguido de una o más letras
- ER para frases que contengan la palabra perro: cero o más letras (a-z) o espacios en blanco, seguido de las letras ‘p’, ‘e’, ‘r’, ‘r’ y ‘o’ (en ese orden), seguido de cero o más letras o espacios en blanco

Expresiones regulares

- .NET Framework incluye la clase Regex para poder crear expresiones regulares y utilizarlas para comprobar que un determinado texto cumpla con ellas

- Se utiliza, por ejemplo, para verificar que una entrada por pantalla es válida. Lo más probable es que los métodos Parse() y TryParse() de int y decimal, o el método ParseExact() de DateTime, utilicen internamente expresiones regulares para devolver su resultado.

Expresiones regulares

Su uso para verificar una entrada por pantalla sería:

- Se escribe una expresión regular empleando la sintaxis definida en la norma POSIX conocida como ERE (Extended Regular Expressions): string reForNumbers = “^[0-9]+$”;
- Se crea un objeto de clase Regex: Regex reNumber = new Regex(reForNumbers);
- Cada vez que se quiera comprobar si un texto cumple con esa ER, se usa el método IsMatch():
  if (reNumber.IsMatch(Console.ReadLine()) == true) { /* el código que sea… */ }

Expresiones regulares

También se puede usar el método Match() (más adelante se explican los paréntesis):

- ```
  string reForThreeNumbers = "^([0-9]+),([0-9]+),([0-9]+)$";
  Regex re = new Regex(reForThreeNumbers);

  Match matchFor3Nums = re.Match("12,34,56");
  if (matchFor3Nums.Success == true) {
      Console.WriteLine($“1º número: {matchFor3Nums.Captures[0].Value}”);
      Console.WriteLine($“2º número: {matchFor3Nums.Captures[1].Value}”);
      Console.WriteLine($“3º número: {matchFor3Nums.Captures[2].Value}”);
  }
  ```
  ```

Expresiones regulares

- Las ER tienen otros usos, como buscar y/o reemplazar todas las veces que se cumple la ER en un texto, pero no se van a ver en el curso

- Enlaces de interés (en inglés, pero con versión en español):
  https://en.wikipedia.org/wiki/Regular_expression
  https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference

Expresiones regulares

La sintaxis Extended Regular Expressions descrita por la norma POSIX, usada para escribir el patrón de las ER, se explica aquí de forma básica (dejando muchos cabos sueltos):

- Carácter '^': usado para representar el inicio del string a comprobar
- Carácter '$': usado para representar el final del string a comprobar
- Carácter '.': carácter comodín. admite cualquier carácter individual en ese punto de la ER
- Carácter '*': usado tras un carácter o una subexpresión, la admite cero o más veces
- Carácter '?': usado tras un carácter o una subexpresión, la admite cero veces o una vez
- Carácter '+': usado tras un carácter o una subexpresión, la admite una o más veces
- Carácter '|': entre dos caracteres o subexpresiones, admite una u otra

Expresiones regulares

La sintaxis Extended Regular Expressions descrita por la norma POSIX, usada para escribir el patrón de las ER, se explica aquí de forma básica (dejando muchos cabos sueltos):

- Caracteres '[' y ']': definen que en ese punto de la ER debe haber uno de los varios caracteres incluidos entre los corchetes.
- Para definir un rango de caracteres, se usa el carácter '-'(1º a izda, último a dcha). Ejemplo: [a-zA-Z] para letra mayúscula o minúscula.
- Para definir que los caracteres válidos de la ER son los que NO están incluidos entre corchetes, se usa ^ dentro de los corchetes. Ejemplo: [^0-9] para NO dígito

Expresiones regulares

La sintaxis Extended Regular Expressions descrita por la norma POSIX, usada para escribir el patrón de las ER, se explica aquí de forma básica (dejando muchos cabos sueltos):

- Caracteres ‘{‘ y ‘}’: Precedidos de carácter o subexpresión, si dentro de los corchetes hay un número, se admite el carácter o subexpresión ese nº exacto de veces.
- Si dentro de los corchetes hay un número y una coma, se admite el carácter o subexpresión anterior ese nº o mas veces
- Si dentro de los corchetes hay dos números separados por coma, se admite el carácter o subexpresión anterior un nº de veces entre el primer y el segundo número

Expresiones regulares

La sintaxis Extended Regular Expressions descrita por la norma POSIX, usada para escribir el patrón de las ER, se explica aquí de forma básica (dejando muchos cabos sueltos):

- Caracteres ‘(‘ y ‘)’: crean una subexpresión con lo que contienen dentro del paréntesis
- Carácter ‘\’ seguido de nº entre 1 y 9: incluye en ese punto de la ER una subexpresión ya definida en algún punto anterior de la ER: para \1 será la 1º, para \2 la segunda, etc… Sirve para reconocer una misma cadena en distintos puntos del texto parseado

06

Ficheros de texto

Ficheros de texto

Namespace: System.IO

- Para comprobar si un string corresponde al path de un fichero existente: string[] **File.Exists(string pathFile)**

```
static void Main(string[] args)
{
    string filePath = "../../ficheros/ficheroTexto.txt";
    if (File.Exists(filePath))
    {
        string text = File.ReadAllText(filePath);
        Console.WriteLine(text);
    }

    Console.ReadKey();
}
```

Ficheros de texto

Namespace: System.IO

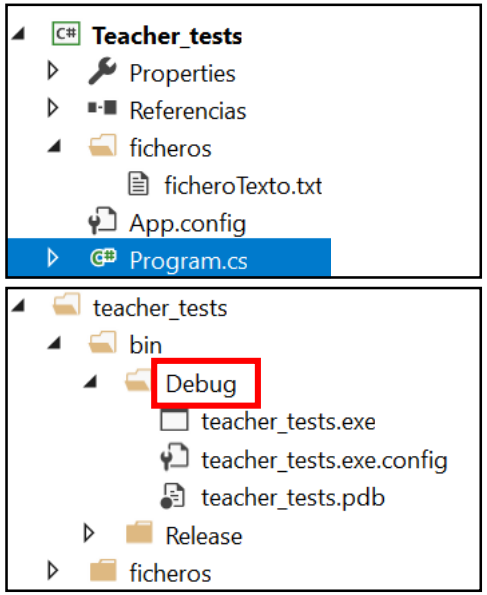
- Para leer TODO el contenido de un fichero de texto y guardarlo en una variable string:
  string **File.ReadAllText(string pathFile)** // pathFile puede ser absoluto o relativo

```
using System;
using System.IO;

namespace Program
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = File.ReadAllText("../../ficheros/ficheroTexto.txt");
            Console.WriteLine(text);

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

Debug | Any CPU | Iniciar



Ficheros de texto

Namespace: System.IO

- Para leer todas las líneas de un fichero de texto y guardarlas en un array de string: string[] **File.ReadAllLines(string pathFile)**

```
static void Main(string[] args)
{
    string[] lines = File.ReadAllLines("../../ficheros/ficheroTexto.txt");

    foreach (string line in lines)
    {
        Console.WriteLine(line);
    }

    Console.ReadKey();
}
```

Ficheros de texto

Namespace: System.IO

- Para ir leyendo poco a poco, controlando la posición del cursor de lectura, se emplea la clase **StreamReader**

```
static void Main(string[] args)
{
    StreamReader file = new StreamReader("../../ficheros/ficheroTexto.txt");
    int numLineasLeidas = 0; string lineaActual; bool finFichero = false;

    do
    {
        lineaActual = file.ReadLine();
        if (lineaActual == null) { finFichero = true; }
        else
        {
            Console.WriteLine(lineaActual);
            numLineasLeidas++;
        }
    } while (finFichero == false);

    file.Close();
    Console.WriteLine("El fichero tiene " + numLineasLeidas + " líneas");

    Console.ReadKey();
}
```

Ficheros de texto

Namespace: System.IO

- Para ir leyendo poco a poco, controlando la posición del cursor de lectura, se emplea la clase StreamReader. Hay 3 métodos para leer de un StreamReader:

  string **ReadLine**() -> devuelve una línea del fichero y salta a la siguiente

  int **Read**() -> devuelve el código ASCII del siguiente carácter del fichero, y avanza el cursor una posición

  int **Read(char[] buffer, int index, int count)** -> considerando la posición actual del cursor como índice 0, empieza en el carácter que se encuentra {index} caracteres más adelante del cursor, y desde ese primer carácter, guarda en buffer los {count} caracteres siguientes. Y NO AVANZA LA POSICIÓN DEL CURSOR

Ficheros de texto

Namespace: System.IO

- Para ir leyendo poco a poco, controlando la posición del cursor de lectura, se emplea la clase StreamReader. Al acabar de utilizar un StreamReader, hay que cerrarlo:

  void **Close**() -> cierra el cursor de lectura, liberando los distintos recursos que tiene.

Ficheros de texto

Namespace: System.IO

- Para escribir un string completo en un fichero:
  void **File.WriteAllText(string pathFile, string text)**

```
static void Main(string[] args)
{
    string filePath = "../../ficheros/ficheroTexto.txt";
    string text = "Aquí escribo un texto largo " +
                  "que pretendo que acabe siendo " +
                  "el contenido completo de un " +
                  "fichero de texto determinado. ¡Gracias!";
    File.WriteAllText(filePath, text);

    Console.ReadKey();
}
```

Ficheros de texto

Namespace: System.IO

- Para escribir en distintas líneas de un fichero los distintos strings de un array de string:
  void **File.WriteAllLines(string pathFile, string[] lines)**

```
static void Main(string[] args)
{
    string filePath = "../../ficheros/ficheroTexto.txt";
    string[] lines = new string[3]
    {
        "Una línea",
        "Otra línea",
        "¡y otra más!"
    };
    File.WriteAllLines(filePath, lines);

    Console.ReadKey();
}
```

Ficheros de texto

Namespace: System.IO

- Para escribiendo poco a poco, se usa la clase **StreamWriter**

```
static void Main(string[] args)
{
    string filePath = "../../ficheros/ficheroTexto.txt";
    string[] lineas = new string[3]
    {
        "Una línea", "Otra línea", "¡y otra más!"
    };
    StreamWriter file = new StreamWriter(filePath);
    foreach (string linea in lineas)
    {
        if (!linea.Contains("Otra"))
        {
            file.WriteLine(linea);
        }
    }

    Console.ReadKey();
}
```

Ficheros de texto

Namespace: System.IO

- Si se crea un nuevo StreamWriter con un segundo parámetro a true, las líneas que se añadan al fichero se añadirán al final, después de lo que ya hubiera escrito de antes.

  Sin ese parámetro a true, se borrará todo el texto del fichero que hubiera escrito de antes, y las líneas que se escriban se escribirán desde el principio del fichero.

```
static void Main(string[] args)
{
    string filePath = "../../ficheros/ficheroTexto.txt";
    string[] lineas = new string[3]
    {
        "Una línea", "Otra línea", "¡y otra más!"
    };
    StreamWriter file = new StreamWriter(filePath);
    foreach (string linea in lineas)
    {
        if (!linea.Contains("Otra"))
        {
            file.WriteLine(linea);
        }
    }

    file = new StreamWriter(filePath, true);
    file.WriteLine("Soy una línea nueva");

    Console.ReadKey();
}
```

Ficheros binarios

Namespace: System.IO

- Un fichero binario es un fichero que no solo contiene 0s y 1s que codifican caracteres imprimibles, sino que puede contener0s y 1s que codifican números (enteros o decimales), valores booleanos, e incluso objetos (instancias) de una determinada clase

Ficheros binarios

Namespace: System.IO

- Para escribir elementos en un fichero binario, intervienen dos clases: FileStream y BinaryWriter

  Hay que llamar al método Open() de FileStream con el 2º parámetro indicando modo de creación de fichero (FileMode.Create) o modo adición (FileMode.Append), según queramos sobreescribir el fichero o añadir elementos al final.

  La clase BinaryWriter tiene métodos Read({tipo}) para todos los tipos de datos primitivos (byte, int, long, float, double, decimal, char, string, bool).

```
static void Main(string[] args)
{
    string filePath = "../../ficheros/ficheroTexto.txt";
    FileStream fs = File.Open(filePath, FileMode.Create);
    BinaryWriter writer = new BinaryWriter(fs);
    writer.Write(1.250F);
    writer.Write("c:/Temp");
    writer.Write(10);
    writer.Write(true);

    Console.ReadKey();
}
```

Ficheros binarios

Namespace: System.IO

- Para leer elementos de un fichero binario, intervienen dos clases: FileStream y BinaryReader

Hay que llamar al método Open() de FileStream con el 2º parámetro indicando modo de lectura de fichero (FileMode.Open).

La clase BinaryReader tiene métodos Read{tipo}() diferentes para cada tipo de dato primitivo (ReadByte, ReadInt32, ReadInt64, ReadSingle, ReadDouble, ReadDecimal, ReadChar, ReadString, ReadBool).

```
static void Main(string[] args)
{
    string filePath = "../../ficheros/ficheroTexto.txt";
    FileStream fs = File.Open(filePath, FileMode.Open);

    if (File.Exists(filePath)) {
        BinaryReader reader = new BinaryReader(fs);

        float aspectRatio = reader.ReadSingle();
        string tempDirectory = reader.ReadString();
        int autoSaveTime = reader.ReadInt32();
        bool showStatusBar = reader.ReadBoolean();
    }

    Console.ReadKey();
}
```

07

Convención de nombres

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do use **PascalCasing** for class names and method names.

```
1.  public class ClientActivity
2.  {
3.      public void ClearStatistics()
4.      {
5.          //...
6.      }
7.      public void CalculateStatistics()
8.      {
9.          //...
10.     }
11. }
```

Why: consistent with the Microsoft's .NET Framework and easy to read.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do use **camelCasing** for local variables and method arguments.

```
1.  public class UserLog
2.  {
3.      public void Add(LogEvent logEvent)
4.      {
5.          int itemCount = logEvent.Items.Count;
6.          // ...
7.      }
8.  }
```

Why: consistent with the Microsoft's .NET Framework and easy to read.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do not use **Hungarian** notation or any other type identification in identifiers

```
1.  // Correct
2.  int counter;
3.  string name;
4.
5.  // Avoid
6.  int iCounter;
7.  string strName;
```

Why: consistent with the Microsoft's .NET Framework and Visual Studio IDE makes determining types very easy (via tooltips). In general you want to avoid type indicators in any identifier.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do not use **Screaming Caps** for constants or readonly variables

```
1.  // Correct
2.  public static const string ShippingType = "DropShip";
3.
4.  // Avoid
5.  public static const string SHIPPINGTYPE = "DropShip";
```

Why: consistent with the Microsoft's .NET Framework. Caps grap too much attention.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

avoid using **Abbreviations**. Exceptions: abbreviations commonly used as names, such as **Id, Xml, Ftp, Uri**

```
1.  // Correct
2.  UserGroup userGroup;
3.  Assignment employeeAssignment;
4.
5.  // Avoid
6.  UserGroup usrGrp;
7.  Assignment empAssignment;
8.
9.  // Exceptions
10. CustomerId customerId;
11. XmlDocument xmlDocument;
12. FtpHelper ftpHelper;
13. UriPart uriPart;
```

Why: consistent with the Microsoft's .NET Framework and prevents inconsistent abbreviations.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do use **PascalCasing** for abbreviations 3 characters or more (2 chars are both uppercase)

```
1.  HtmlHelper htmlHelper;
2.  FtpTransfer ftpTransfer;
3.  UIControl uiControl;
```

Why: consistent with the Microsoft's .NET Framework. Caps would grap visually too much attention.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do not use **Underscores** in identifiers. Exception: you can prefix private static variables with an underscore.

```
1.  // Correct
2.  public DateTime clientAppointment;
3.  public TimeSpan timeLeft;
4.
5.  // Avoid
6.  public DateTime client_Appointment;
7.  public TimeSpan time_Left;
8.
9.  // Exception
10. private DateTime _registrationDate;
```

Why: consistent with the Microsoft's .NET Framework and makes code more natural to read (without 'slur'). Also avoids underline stress (inability to see underline).

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do use **predefined type names** instead of system type names like Int16, Single, UInt64, etc

```
1.  // Correct
2.  string firstName;
3.  int lastIndex;
4.  bool isSaved;
5.
6.  // Avoid
7.  String firstName;
8.  Int32 lastIndex;
9.  Boolean isSaved;
```

Why: consistent with the Microsoft's .NET Framework and makes code more natural to read.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do use implicit type **var** for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.

```
1.  var stream = File.Create(path);
2.  var customers = new Dictionary();
3.
4.  // Exceptions
5.  int index = 100;
6.  string timeSheet;
7.  bool isCompleted;
```

Why: removes clutter, particularly with complex generic types. Type is easily detected with Visual Studio tooltips.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do use noun or noun phrases to name a class.

```
1.  public class Employee
2.  {
3.  }
4.  public class BusinessLocation
5.  {
6.  }
7.  public class DocumentCollection
8.  {
9.  }
```

Why: consistent with the Microsoft's .NET Framework and easy to remember.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do prefix interfaces with the letter `I`. Interface names are noun (phrases) or adjectives.

```
1.  public interface IShape
2.  {
3.  }
4.  public interface IShapeCollection
5.  {
6.  }
7.  public interface IGroupable
8.  {
9.  }
```

Why: consistent with the Microsoft's .NET Framework.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.

```
1.  // Located in Task.cs
2.  public partial class Task
3.  {
4.      //...
5.  }
```

```
1.  // Located in Task.generated.cs
2.  public partial class Task
3.  {
4.      //...
5.  }
```

Why: consistent with the Microsoft practices. Files are alphabetically sorted and partial classes remain adjacent.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do organize namespaces with a clearly defined structure

```
1.  // Examples
2.  namespace Company.Product.Module.SubModule
3.  namespace Product.Module.Component
4.  namespace Product.Layer.Module.Group
```

Why: consistent with the Microsoft's .NET Framework. Maintains good organization of your code base.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do vertically align curly brackets.

```
1.  // Correct
2.  class Program
3.  {
4.      static void Main(string[] args)
5.      {
6.      }
7.  }
```

Why: Microsoft has a different standard, but developers have overwhelmingly preferred vertically aligned brackets.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do declare all member variables at the top of a class, with static variables at the very top.

```
1.  // Correct
2.  public class Account
3.  {
4.      public static string BankName;
5.      public static decimal Reserves;
6.
7.      public string Number {get; set;}
8.      public DateTime DateOpened {get; set;}
9.      public DateTime DateClosed {get; set;}
10.     public decimal Balance {get; set;}
11.
12.     // Constructor
13.     public Account()
14.     {
15.         // ...
16.     }
17. }
```

Why: generally accepted practice that prevents the need to hunt for variable declarations.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do use singular names for enums. Exception: bit field enums.

```
1.  // Correct
2.  public enum Color
3.  {
4.      Red,
5.      Green,
6.      Blue,
7.      Yellow,
8.      Magenta,
9.      Cyan
10. }
11.
12. // Exception
13. [Flags]
14. public enum Dockings
15. {
16.     None = 0,
17.     Top = 1,
18.     Right = 2,
19.     Bottom = 4,
20.     Left = 8
21. }
```

Why: consistent with the Microsoft's .NET Framework and makes the code more natural to read. Plural flags because enum can hold multiple values (using bitwise 'OR').

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do not explicitly specify a type of an enum or values of enums (except bit fields)

```
1.  // Don't
2.  public enum Direction : long
3.  {
4.      North = 1,
5.      East = 2,
6.      South = 3,
7.      West = 4
8.  }
9.
10. // Correct
11. public enum Direction
12. {
13.     North,
14.     East,
15.     South,
16.     West
17. }
```

Why: can create confusion when relying on actual types and values.

Convención de nombres

https://www.dofactory.com/reference/csharp-coding-standards

do not suffix enum names with Enum

```
1.  // Don't
2.  public enum CoinEnum
3.  {
4.      Penny,
5.      Nickel,
6.      Dime,
7.      Quarter,
8.      Dollar
9.  }
10.
11. // Correct
12. public enum Coin
13. {
14.     Penny,
15.     Nickel,
16.     Dime,
17.     Quarter,
18.     Dollar
19. }
```

Why: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers.

MUCHAS GRACIAS

Pasión por la innovación

www.integratecnologia.es