Tema 3 – Otros aspectos de C#

Arrays Unidimensionales

La forma de trabajo con arrays es muy similar a la hecha en Java. Veamos un par de ejemplos de repaso de creación y visualización de arrays.

```
int x, i;
int[] v;

Console.Write("Introduce cantidad de elementos del vector: ");
x = Convert.ToInt32(Console.ReadLine());
v = new int[x];
Console.WriteLine("El tamaño de V es {0}", v.Length);

Console.WriteLine("introduce los valores");
for (i = 0; i < x; i++)
{
    v[i] = Convert.ToInt32(Console.ReadLine());
}
for (i = 0; i < x; i++)
{
    Console.WriteLine("Posición: {0}\tValor: {1}", i, v[i]);
}
```

También se le puede dar una inicialización de valores por defecto:

```
int[] tabla = { 5, 1, 4, 0 };
```

Lenght: Propiedad que devuelve el número total de elementos que tiene un array.

Existe la clase **Array** con métodos estáticos para realizar ciertas tareas. En principio no los usaremos.

Si se quiere eliminar totalmente un array basta con asignarlo a null.

Multidimensionales

En este caso los cambios más notables son el uso de la coma para indicar múltiples dimensiones (en lugar de abrir y cerrar corchetes) y por otro lado el uso de otras propiedades para llegar a los extremos de cada dirección. Veamos un ejemplo aclaratorio y luego describimos las propiedades y métodos implicados:

```
int i, j;
int[,] bi = new int[3, 5];
Random g = new Random();

Console.WriteLine("Array bidimensional");
for (i = 0; i < bi.GetLength(0); i++)       // Cantidad de filas
{
    for (j = 0; j < bi.GetLength(1); j++)   // Cantidad de columnas
    {
        bi[i, j] = g.Next(1,21);             // Damos valores iniciales
    }
}
for (i = 0; i <= bi.GetUpperBound(0); i++)     // Limite superior de filas
{
    for (j = 0; j <= bi.GetUpperBound(1); j++) //Limite superior de columnas
    {
        Console.Write("{0,3}", bi[i, j]);
    }
    Console.WriteLine();
}
Console.WriteLine("El nº de dimensiones del array es {0}", bi.Rank);
```

Propiedades y métodos usados:

Length*:* Sigue siendo el número total de elementos, no sirve para recorrer el array (En realidad se podría hacer con un único bucle pero gestionando las coordenadas en variables aparte del contador del for).

GetLenght*(Dimension)*: Devuelve el número de elementos en cada dimensión. Empieza en 0 que en el ejemplo anterior indica el número de filas mientras que 1 es el número de columnas. Corresponde al valor puesto en la inicialización del vector. Lo usaremos para recorrer el array desde 0 hasta el valor **menor que** *GetLength.*

GetUpperBound*(Dimensión)*: Devuelve el índice del último elemento de la

Dimensión. También se puede usar para recorrer teniendo en cuenta que iremos desde 0 hasta el valor **menor o igual que** *GetUpperBound*.

Rank: Rango del array (Número de dimensiones).

Si quiero inicializarlo con valores predeterminados lo hago igual que en Java:

```
int[,] dosDimensiones = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 } };
```

Tablas dentadas

Existe en C# la posibilidad de manejar los arrays bidimensionales exactamente igual a como se hace en Java: usando dobles corchetes en la inicialización y sólo el **Length** en los extremos del recorrido, pero tiene un objetivo muy concreto y es el de hacer arrays bidimensionales cuyas filas tienen distintas longitudes: es lo que se denomina tabla dentada. Se ve con un ejemplo que, acostumbrados al Java, su comprensión no debe tener dificultad ninguna:

```
Random g = new Random();
int[][] tabla = new int[3][];
tabla[0] = new int[4];
tabla[1] = new int[2];
tabla[2] = new int[5];

for (int i = 0; i < tabla.Length; i++)
{
    for (int j = 0; j < tabla[i].Length; j++)
    {
        tabla[i][j] = g.Next(10, 40);
    }

}
for (int i = 0; i < tabla.Length; i++)
{
    for (int j = 0; j < tabla[i].Length; j++)
    {
        Console.Write("{0,3}", tabla[i][j]);
    }
    Console.WriteLine();
}
```

Cadenas (string)

En C# las cadenas se establece mediante el tipo **string**. El uso es muy similar a Java, basándonos principalmente en funciones que manejan cadenas y el operador de concatenación. Como el resto de los tipos, también son objetos sin embargo sobrecargan dos operadores para que su uso sea más intuitivo:

El **operador +** se usa para la concatenación de cadenas o de caracteres y cadenas. Cuando se concatena con otro tipo, implícitamente se está llamando al método **ToString**().

El **operador ==** compara el contenido de las cadenas, no las direcciones de memoria. Por lo que podemos comparar cadenas directamente en una posición de condición como un **if** o un **while**.:

```
string nombre;
Console.Write("Introduce tu nombre: ");
nombre = Console.ReadLine();
if (nombre == "Curro")
{
    Console.WriteLine("No quiero hablar contigo");
}
else
{
    Console.WriteLine("¡¡¡Bienvenido!!! :-)");
}
```

En el caso de que quiera hacer comprobación sin tener en cuenta mayúsculas o minúsculas puedo usar **ToUpper** o **ToLower**

```
if (nombre.ToUpper()=="CURRO")
```

o **Equals con un segundo parámetro** indicando que ignore mayúsculas y minúsculas, al depender del elemento cultural tiene en cuenta elementos como tildes en el castellano (á y Á serían iguales, o ñ y Ñ):

```
if (nombre.Equals("Curro",StringComparison.CurrentCultureIgnoreCase))
```

Una cadena se comporta igual que un array unidimensional, siendo vectores de caracteres que comienzan en el índice 0. Por tanto podemos acceder a sus caracteres por separado mediante el uso de corchetes. Por ejemplo:

```
Console.WriteLine(nombre[0]);
```

Sin embargo hay que tener en cuenta que los strings son inmutables y por tanto C# no permite la modificación carácter a carácter. Para ello se usarán métodos de la clase String.

Si fuera necesario trabajar a nivel de carácter, se debe usar la clase **Text.StringBuilder,** la cual sí permite usar la propiedad *chars* tanto para lectura como escritura.

Algunas propiedades y métodos de la clase String (y Char).

Remove(int pos, int n) *:* elimina subcadenas de *n* caracteres a partir de pos. Devuelve la cadena modificada.

Insert(int pos, string cad): Inserta *cad* a partir de *pos.* Devuelve la cadena modificada.

Replace(Cadena a Sustituir, sustituta): Devuelve la cadena resultante de sustituir en la cadena sobre la que se aplica toda aparición de la cadena a sustituir indicada por la cadena sustituta especificada como segundo parámetro.

Lenght: Devuelve la longitud de la cadena.

EndsWith(cadena): Función booleana. Comprueba si la cadena acaba por el parámetro

StartsWith(cadena): Función booleana. Comprueba si la cadena empieza por el parámetro

Substring(posición inicial, longitud): Función que devuelve una subcadena.

Split(array de char): devuelve un array de cadenas tras "romper" la original por los caracteres indicados en el array.

ToUpper, ToLower: Funciones que devuelven el string pasado a mayúsculas y minúsculas respectivamente. (Tiene en cuenta configuración regional).

Trim(): Función que quita espacios al principio y final de la cadena y la devuelve. Existe *TrimEnd* y *TrimStart* para quitar los espacios solo del final o del principio respectivamente. Devuelve la cadena modificada.

Funciones estáticas:

string.Format(cadena): da formateo igual que se vio para WriteLine pero devolviendo la cadena formateada.

string.IsNullOrEmpty(cadena): Comprubea si el string es null o es una cadena vacía "" .

Char.IsDigit(char c): Comprueba si c es un dígito numérico

Char.IsLetter(char c): Comprueba si c es una letra.

Char.IsPunctuation(char c): Comprueba si c es un signo de puntuación.

Como apunte final, indicar que como en C# existe la sobrecarga de operadores, **a la hora de comparar cadenas es mejor hacerlo con == que con Equals.** Por un lado un motivo de claridad, pues la simbología es más estándar. Por otro hay un motivo de detección de fallos por tipos ya que Equals al aceptar object puede dar más problemas. Veamos un ejemplo:

Si estamos en un caso en que comparamos por ejemplo la primera letra de un string con un carácter. Tenemos las siguientes variables:

```
letra="C"
palabra="CAFE"
```

Si hacemos:

```
palabra[0].Equals(letra)
```

resulta que parece que funciona pero da **false** siempre ya que comparamos **char** con **string** y es posible que no nos demos cuenta. Sin embargo si

hacemos:

```
palabra[0]==letra
```

ya no compila y nos informa del error pues solo se permite el uso del == entre strings (o entre caracteres).

En los siguientes enlaces más información:
https://docs.microsoft.com/en-us/dotnet/api/system.string?view=netframework-4.7.2
https://docs.microsoft.com/en-us/dotnet/api/system.char?view=netframework-4.7.2

Cadenas interpoladas

El uso de la variable en el WriteLine directamente entre llave es lo que se denominan cadenas interpoladas a diferencia del formato que vimos hasta ahora de cadenas compuestas. Están permitidas desde C# 6.0 y por supuesto pueden ser usadas con la consola o en modo gráfico. Se trata de anteponer el $ a la cadena y puedes insertar la variable o constante con formato entre las llaves. Ojo, no es necesario string.Format.

Veamos un ejemplo:

```
int[] v  = { 5, 1, 4, 0 };
DateTime date = DateTime.Now;

//Uso var para evitar el using y una declaración excesivamente larga
var culture = new System.Globalization.CultureInfo("es-ES");
String day = culture.DateTimeFormat.GetDayName(date.DayOfWeek);

Console.WriteLine($"¡Hola! Hoy es {day}, y son las {date:HH:mm}.");
Console.WriteLine("Veamos el contenido de un vector");
for (int i = 0; i < v.Length; i++)
{
    Console.WriteLine($"Posición: {i,3}\tValor: {v[i],3}");
}
```

Puedes leer más en :

https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/tokens/interpolated

ToString

El método ToString es de sobra conocido. Sin embargo conviene indicar que dispone de una serie de sobrecargas per permite hacer la conversión a distintos formatos. Sería similar a usar un string.Format pero ya directamente en el ToString. Algunos ejemplos:

```
int num = 60;

// 3 decimales
Console.WriteLine((num/7.0).ToString("F3"));

// Hexadecimal con 4 dígitos
Console.WriteLine(num.ToString("X4"));

// Para otras bases se usa Convert.ToString
Console.WriteLine(Convert.ToString(num, 2)); //Binario
Console.WriteLine(Convert.ToString(num, 8)); //Octal

// Formatos según configuración regional
Console.WriteLine(DateTime.Now.ToString(
    CultureInfo.CurrentCulture.DateTimeFormat.LongDatePattern));

Console.WriteLine(DateTime.Now.ToString(
    CultureInfo.CurrentCulture.DateTimeFormat.ShortDatePattern));
```

Puedes leer más en :

https://learn.microsoft.com/es-es/dotnet/standard/base-types/standard-numeric-format-strings

Colecciones

Al igual que en Java, C# dispone de varios tipos de colecciones. En este apartado nos limitaremos a usar la colección genérica **List<T>.** El *List<T>* de C# es el similar al *ArrayList<T>* de Java, el cual permite especificar el tipo de objeto que vamos a incluir en cada posición de la colección.

No se debe confundir con el *ArrayList* de C#, que no es tipado y permite meter *object* por lo que no se especifica el tipo de colección (sería similar a List<object>).

La clase **ArrayList** se encuentra dentro de *System.Collections* y **List<T>** en *System.Collections.Generic.*

En principio vamos a trabajar sólo con este tipo de colecciones para aprender su manejo, pero a lo largo del curso nos iremos encontrando otras con funcionamiento muy similar aunque con puntualizaciones que habrá que leer en la documentación.

A continuación indicamos algunos métodos y propiedades de estas colecciones:

Count: Cantidad de elementos.

Add(elemento): Añadir un elemento (al final de la colección).

RemoveAt(posición entre 0 y Count-1): Eliminar un elemento en la posición indicada

Remove(objeto): Para eliminar cierto objeto.

Insert(nº de posición, elemento): Insertar un elemento en la posición indicada

Clear(): Borra el contenido de una colección

Contains(elemento): Devuelve true si el elemento se encuentra en la colección

IndexOf(elemento): Devuelve el índice del elemento (la primera vez que aparece)

Sort: Ordena la colección. Ojo, da error si la colección contiene objetos de distintos tipos.

Bucle foreach

Para recorrer los elementos de una colección se puede usar un bucle *for* clásico de forma equivalente al recorrido de un array o se puede usar un bucle similar al *for* mejorado de Java denominado *foreach*.

Formato genérico:

```
foreach (tipo_elemento nombre_elemento in nombre_colección) {
        Operaciones
}
```

Este bucle es una forma muy cómoda de recorrer a la vez que se hace el casting. La limitación es que la variable del *foreach* es de solo lectura, no sirve, por tanto, para modificar el contenido de la colección.

Por supuesto el bucle *foreach* puede ser usado también con arrays, no solo con colecciones.

Ejemplo:

```
using System;
using System.Collections;
using System.Collections.Generic;

class Program
{
    //Muestra la colección de enteros con foreach
    static void muestraColInt(List<int> col)
    {
        foreach (int num in col)
        {
            Console.Write("{0,5}", num);
        }
        Console.WriteLine();
    }
```

```
// Muestra la colección de objetos con foreach
static void muestraColObject(ArrayList col)
{
    foreach (string s in col)
    {
        Console.WriteLine(s);
    }
    Console.WriteLine();
}

static void Main(string[] args)
{
    //Iniciamos las colecciones
    ArrayList colObjetos = new ArrayList();
    List<int> colEnteros = new List<int>();
    Random g = new Random();

    //Rellenamos colección de enteros
    for (int i = 0; i < 5; i++)
    {
        colEnteros.Add(g.Next(0, 10));
    }

    //Rellenamos la colección de objetos sólo con strings
    colObjetos.Add("Isaac Asimov");
    colObjetos.Add("Robert A. Heinlein");
    colObjetos.Add("Arthur C. Clarke");
    colObjetos.Add("Connie Willis");

    //Mostramos las colecciones
    muestraColInt(colEnteros);
    muestraColObject(colObjetos);

    //Ordenamos las colecciones
    colEnteros.Sort();
    colObjetos.Sort();

    //Modificamos las colecciones usando un for clásico
    for (int i = 0; i < colEnteros.Count; i++)
    {
        colEnteros[i]++;
    }
    Console.WriteLine();
```

```
        for (int i = 0; i < colObjetos.Count; i++)
        {
            colObjetos[i] = $"{(i + 1)}.- {colObjetos[i]}";
        }

        // Volvemos a mostrar
        muestraColInt(colEnteros);
        muestraColObject(colObjetos);

        //Eliminamos elementos de las colección y la mostramos
        colObjetos.RemoveAt(0);
        colObjetos.Remove("3.- Isaac Asimov");

        // Volvemos a mostrar
        muestraColObject(colObjetos);

        //Puedo introducir cualquier tipo en el ArrayList.
        colObjetos.Add(231);
        colObjetos.Add(Math.PI);

        // Si ahora quiero recorrer solo un tipo
        // hay que tener en cuenta el polimorfismo
        foreach (object s in colObjetos)
        {
            if (!(s is string))
            {
                Console.WriteLine($"{s,6:F2}");
            }
        }
        Console.ReadKey();
    }
}
```

Tablas Hash (diccionario)

Es una colección especial en la cual el índice en lugar de ser numérico es un *string* de forma que estas colecciones se componen de pares clave/valor. De hecho se puede ver como dos colecciones interrelacionadas, una de claves y otra de valores asociados a cada clave.

Se usa la clase *Hashtable* como se ve en el ejemplo:

```
Hashtable edades = new Hashtable();

// Introducir elementos (calve, valor)
edades.Add("Ana", 20);
edades.Add("Juan", 31);
edades.Add("Pablo", 2);
edades.Add("Maria", 12);

// Mostrar un elemento mediante hashing
Console.WriteLine("Ana tiene {0} años", edades["Ana"]);

//Recorrer la colección pares clave/valor
foreach (DictionaryEntry de in edades)
{
    Console.WriteLine("{0} tiene {1} años", de.Key, de.Value);
}

//Recorrer la colección solo de valores
foreach (int de in edades.Values)
{
    Console.WriteLine(de);
}

//Recorrer la colección solo de claves
foreach (string de in edades.Keys)
{
    Console.WriteLine(de);
}
```

Existe una implementación tipada denominada **Dictionary** de funcionamiento muy similar. Prueba a cambiar la declaración de Hashtable por:

```
Dictionary<String, int> edades = new Dictionary<string, int>();
```

Y funciona igual salvo el foreach primero en el que debes cambiar el tipo DictionaryEntry por KeyValuePairs de esta manera:

```
foreach (KeyValuePair<string, int> de in edades)
{
    Console.WriteLine("{0} tiene {1} años", de.Key, de.Value);
}
```

Control de excepciones

Respecto a lo que necesitamos podemos ver el control de excepciones prácticamente idéntico al de Java usando la estructura **try/catch/finally.** Evidentemente van a cambiar los nombres de las clases de excepción, pero en cualquier caso todas heredan de la clase base **Exception**.

La diferencia quizá más notable es que en caso de no necesitar usar el objeto excepción, no es necesario indicarlo. Es más, si queremos capturar una excepción genérica después del *catch* no es necesario poner nada.

Otra diferencia es que en C# **no es necesario** el uso de la cláusula **throws** en las cabeceras de las funciones.

Para l**anzar una excepción** se realiza también mediante el comando **throw**.

Ejemplo:

```
int num;
bool error = false;

do
{
    //Planteamos un bucle para volver a ejecutar la opción
    try
    {
        error = false;
        Console.WriteLine("Por favor, introduce un número positivo");
        num = Convert.ToInt32(Console.ReadLine());
        if (num < 0)
               throw new System.ArgumentOutOfRangeException();

        //Mediante checked se controlan desbordamientos en operaciones
        checked
        {
               Console.WriteLine("El cuadrado de {0} es {1}", num, num * num);
        }
    }//Fin del try
    catch (System.FormatException)
    {
        Console.WriteLine("Ha introducido un valor no numérico o no entero");
        error = true;
    }
```

```
        catch (System.OverflowException)
        {
            Console.WriteLine("Número demasiado grande para obtener su cuadrado");
            error = true;
        }

        catch (System.ArgumentOutOfRangeException)
        {
            Console.WriteLine("Se ha pedido un número positivo");
            error = true;
        }

        catch (Exception e)
        {
            Console.WriteLine("Error del tipo:\n\t{0}", e.ToString());
            Environment.Exit(-1);
        }
    } while (error);
```

Aclaraciones:

- La operación **num*num** para números muy grandes puede provocar un desbordamiento. Por defecto en C# es una operación **unchecked**, es decir, que no se controla el desbordamiento. Para forzar el control se usa el comando **checked**. Prueba a quitarlo e introduce como dato 1000000, verás que no salta la excepción.

- En el último *catch,* si no fuera necesario el uso del objeto **Exception e**, se puede obviar. Podría ponerse de la siguiente forma:

```
        catch
        {
            Console.WriteLine("Error indeterminado");
            Environment.Exit(-1);
        }
```

- Mediante *Environment.Exit()* salgo de la aplicación. El número que se le pone entre paréntesis es un código de finalización que podría utilizar el SO u otro programa que ejecute este para saber si la ejecución ha sido correcta (veremos más en el apéndice y en el tema de *threading*).

- Si una vez capturada una excepción tras ejecutar cierto código se desea volver a lanzarla llega con usar la clausula **throw;** Ojo porque el snippet de try mete siempre este elemento y muchs veces no es necesario.

```
catch (System.FormatException)
{
    Console.WriteLine("Ha introducido un valor no numérico o no entero");
    error = true;
    throw;
}
```

- Existe la posibilidad de recoger la excepción de forma condicional mediante la sentencia **when**.

```
catch (System.ArgumentOutOfRangeException) when (num < 10000)
{
    Console.WriteLine("Se ha pedido un número positivo");
    error = true;
}
```

Muy útil para coger varias excepciones que hacen lo mismo:

```
catch (Exception ex) when ( ex is FormatException || ex is … || ex is …)
```

Propiedades principales de la clase Exception (y derivadas)

- *Message*. Descripción del error.
- *Source*. Nombre del objeto o aplicación que provocó el error.
- *StackTrace*. Ruta o traza del código en la que se produjo el error.

TryParse

Los tipos numéricos (int, double, long,…) disponen de una función estática de conversión de *string* a dicho tipo numérico que evita el uso de excepciones, pues ya lo incluye. El formato es:

```
bool tipo.TryParse(string s, out tipo numero);
```

Esta función devuelve *true* si se ha podido hacer la conversión de ***string*** a ***tipo*** y

false en caso contrario. Además si la conversión se realiza, esta se guarda en la variable ***numero***.

Veamos un ejemplo:

```
int n;
bool bandera;
do
{
    Console.WriteLine("Dime un nº");
    bandera = int.TryParse(Console.ReadLine(), out n);
    if (!bandera)
    {
        Console.WriteLine("Eso no es un nº entero, vuelve a intentarlo");
    }
} while (!bandera);

Console.WriteLine($"¡Bien! tu número introducido es:{n,2}");
```

Enumeraciones

Tienen una gran similitud con los enumerados del Java, pero son quizá más sencillos de usar.

Es un tipo de dato creado por nosotros y que contiene las constantes que el programador desee.

Sin embargo, en C# las enumeraciones son constantes numéricas tipo *int* (u otro entero como veremos) pero que las vamos a poder nombrar de una forma más clara.

La forma genérica de crear una enumeración es:

```
enum Nombre : tipo_base {
      constantes
}
```

Puede ser declarada fuera o dentro de una clase. Si se desea que en lugar de *int* sea otro tipo (*byte, short, long*, etc...) se especifica en tipo_base. Si no se indica el tipo base, este se toma *int*.

Veamos un ejemplo

```
enum Dias
{
    Lunes,
    Martes,
    Miercoles,
    Jueves,
    Viernes,
    Sabado,
    Domingo
}
```

Realmente el *Lunes* de esta enumeración es equivalente al número 0 y el *Martes* al número 1,...

Si queremos establecer un número de comienzo distinto al cero lo indicamos asignándole a la primera variable el valor por el que queremos que comience. Por ejemplo:

```
enum Dias
{
    Lunes = 1,
    Martes,
    Miercoles,
    Jueves,
    Viernes,
    Sabado,
    Domingo
}
```

En este caso *Lunes* vale 1 y *Martes* vale 2, etc...

Se puede tener cada constante con un valor e incluso repetir valores según las constantes. Veamos un ejemplo:

```
enum Bancos
{
      Abanca = 2080,
      Bankia = 3922,
      BBVA = 1020,
      Bankinter = 3021,
      BSCH = 4231,
      Banesto = 3489,
      Caixavigo = Abanca
}
```

El problema que se tiene con enumeraciones con el entero repetido es que puede haber conflicto si se pasa de número a enumerado, debe usarse con cuidado.

Para usar las enumeraciones:

```
Dias dia;
dia = Dias.Miercoles;
Console.WriteLine(dia);

//Como número
Console.WriteLine((int)dia);
Console.WriteLine(dia.ToString("D")); // decimal
```

También sería posible usar el *String.Format,* las cadenas interpoladas o el formato de *WriteLine* .

Para asignar directamente un nº a un enumerado debe hacerse mediante casting:

```
dia = (Dias)3;
```

Una enumeración por defecto es estática, por lo que no es necesario instanciar la clase a la que pertenece (si pertenece a alguna) para usarla.

Por ser enteros, le puedo aplicar operadores. Por ejemplo, podría escribir las siguientes sentencias:

```
dia++;

Console.WriteLine(dia);
```

Mostrando **Jueves** como resultado.

Algunos métodos estáticos de la clase Enum:

- **Enum.GetNames()**, devuelve un **string[]** con los nombres de todos los literales que tengan el valor indicado ordenados según su orden de definición en la enumeración. Puede ser útil por ejemplo para plantear un menú:

```
string[] dias = Enum.GetNames(typeof(Dias));
int i = 0;
foreach(string d in dias)
{
      Console.WriteLine($"{++i} .- {d}") ;
}
```

- **static bool isDefined (Type enum, object valor)**: Devuelve un booleano que indica si algún literal de la enumeración indicada tiene el valor indicado.

- **static object Parse(Type enum, string nome, [bool mayusculas])**: Para **convertir una cadena a enumerado** si hay correspondencia. Muy práctica para guardar enumerados como cadenas en archivos, bases de datos o en componentes como ComboBoxes.

  En caso de que no se pueda convertir salta excepción.

  OJO: No es recomendable usarlo para pedir enumerados al usuario, pues debería escribirlo completo, al usuario siempre un menú o combobox en modo gráfico.

Type enum: Tipo del enumerado. Se usa la función typeof.

string nombre: Cadena que representa al valor enumerado.

[bool mayusculas]: Opcional, indica si se desea tener en cuenta o no las capitalización. Si se pone true da igual que la cadena este´con aimayúsculas o minúsculas, si se pone false sí importa.

```
dia = (Dias)Enum.Parse(typeof(Dias), "Viernes", true);

Console.WriteLine(dia);
```

Ejercicios

Nota: A partir de este tema **el control de excepciones** va implícito en la realización de **todos los ejercicios.**

Ejercicio 1

Crear una tabla hash (Hashtable o Dictionary) usando como clave las IPs de ordenadores y como valor la cantidad de memoria RAM que tiene el equipo en GB. Se plantea un menú de introducción de datos, elimina un dato (por clave), muestra de la colección entera y muestra de un elemento de la colección.

Al pedir datos se debe comprobar que la IP es válida y que la cantidad de RAM es un entero positivo.

Ejercicio 2

En un instituto se desea realizar una serie de estadísticos con las notas de los alumnos. Para ello, y como fase inicial de un futuro proyecto, se pide la realización de un programa de simulación de notas de dicho instituto. Se debe crear para ello una tabla de **enteros** (en la clase **Aula** que se describe más abajo) que será rellenada con notas aleatorias entre 0 y 10.

El usuario dispondrá de un menú (en la clase **Menu** que se describe más abajo) con las opciones:

- Calcular la media de notas de toda la tabla.
- Media de un alumno
- Media de una asignatura
- Visualizar notas de un alumno
- Visualizar notas de una asignatura
- Nota máxima y mínima de un alumno
- Visualizar tabla completa

Los nombres de los alumnos estarán en un vector y los nombres de las asignaturas serán un enumerado con las constantes: Pociones, Quidditch, Criaturas, ArtesOscuras .

Debes estructurar el programa en al menos *tres* clases además del enumerado:

- **Clase Aula** donde se encuentra el array bidimensional de notas **público** y todos los métodos de proceso de datos (obtención de medias, devolución de una fila...). Además:
  - El array será de 4 columnas que corresponden *con* las asignaturas.
  - Tendrá un constructor al que se le pasa como parámetro un vector de strings con los nombres de los alumnos. Este vector establecerá por tanto el número de filas del array bidimensional. Debes también guardar dicho vector en una propiedad. Al guardarlo debes quitar espacios en los extremos de cada nombre y pasarlo todo a mayúsculas.
  - Una sobrecarga del constructor al que se le pasa un único string con los nombres separados por comas. Crea un vector con los nombres y llama al otro constructor.
  - Además de otras funciones de procesado, haz una función única que coloque el máximo y el mínimo de un alumno en parámetros por referencia. No devolverá nada.
  - **No** debe tener interfaz de usuario (Ningún Console).
  - Debe ser una clase indexada (Ver Apéndice III) de forma que con el nombre del objeto se pueda ya acceder a las posiciones del array si se desea.
- Otra clase denominada **Menu** donde está el interfaz de usuario: el menú y otros elementos que necesites.

  Lógicamente tendrá una propiedad tipo Aula para gestionar los datos. Usa la indexación del objeto cuando sea necesario acceder a notas.

  El constructor de *Menu* tendrá un número indeterminado de parámetros (Ver apéndice I) tipo string que serán los nombres de los alumnos.

  Habrá por lo menos una función denominada Inicio, que será la única pública, donde se ejecuta el menú principal. Haz otros métodos y

propiedades para modularizar bien la clase.

- Finalmente en la clase **Program** se crea un objeto del tipo Menu y se lanza su función inicio.

Es fundamental que sea **amigable para el usuario**. Entre otras cosas:

- Las medias se trabajarán siempre con 2 decimales.
- Será robusto ante errores del usuario como salidas de rango.
- La entrada de datos debe ser cómoda, siempre con menús (nada de que tenga que escribir, por ejemplo, un nombre).
- Cuando muestres tablas o notas se deben indicar siempre nombres de asignatura/s, alumno/s, etc.
- Las filas y columnas de las tablas cuando se muestren deben estar siempre correctamente alineadas (usa formateo de cadenas, no tabuladores).
- En general el usuario siempre debe estar siempre bien informado: saber qué información aparece en pantalla, qué error comete, etc..

Ejercicio 3

Retoma las clases Persona, Empleado y Directivo del tema anterior (si no lo acabaste remátalo antes).

Se trata de hacer un programa que gestione una colección de trabajadores en una empresa.

Habrá una clase denominada **GestorPersonas** sin ninguna interfaz de usuario donde estará la colección pública y al menos las siguientes funciones:

- int Posicion(int): se le pasa una edad como parámetro y devuelve la posición donde dicho parámetro es mayor al de la colección (posición de inserción).

- Sobrecargada a la anterior se le pasa como parámetro un string en vez de entero. Devuelve la posición de la primera persona cuyo apellido empiece por dicho string.

  Si no existe, devuelve -1.

- bool Eliminar(int , int): se le pasa una posición mínima y una máxima y elimina entre dichas posiciones todos los datos. El parámetro de mínimo es opcional y si no se pasa será 0. Devuelve true si va todo bien y false si alguno de los parámetros está fuera de rango y no se ha hecho la eliminación.

En otra clase denominada **InterfazUsuario** tendrá un menú que disponga de las siguientes posibilidades:

- Insertar nueva persona (Empleado o Directivo). Se inserta siempre por orden de edad creciente.

- Eliminar personas (Se ha de pedir un rango de posiciones). Muestra los datos de las personas a eliminar y pide confirmación previa.

- Visualizar toda las lista de Personas.

  En cada línea aparecerá formateado el número de posición en la colección (3 caracteres), Nombre (máximo 10 caracteres), Apellidos (Máximo 20 caracteres), una E si es empleado y D si es Directivo. Deben aparecer cabeceras en la parte superior.

  Si el Nombre o los apellidos son muy grandes que los corte y le ponga puntos suspensivos al final. Por ejemplo Nabucodonosor aparecerá como Nabucod… (así ocupa los 10 caracteres).

- Visualización de una persona. Se pide el comienzo del apellido y muestra los datos de la primera persona que corresponda simplemente llamando a la función MostrarDatos. Si además es directivo, llama a ganarPasta con parámetro 1000.

- Salir del programa

La clase InterfazUsuario estará bien modularizada con los métodos y propiedades que consideres. El código del propio menú estará en un método denominado Inicio.

Al ejecutar el programa se debe inicializar la colección automáticamente al menos con 2 empleados y un directivo (usa una directiva de compilación para controlar esto). El main estará en la clase **Program** y simplemente creará un objeto del tipo InterfazUsuario y llamará a Inicio.

Por supuesto se debe hacer un programa claro para el usuario informando o salvando los posibles errores que se puedan producir..

Ejercicio 4 (Opcional: Ver apéndices del tema)

a) Realiza una función a la que se le pase de forma obligatoria un símbolo de suma o multiplicación y luego una ristra de tamaño indeterminado de números enteros. Debe realizar la cuenta con todos los números y devolver el resultado (como función, no por pantalla). Si el símbolo de operación que se le pasa es incorrecto, no hará nada.

Realiza un programa que pruebe la función anterior (no es necesario realizar interfaz con el usuario, se pueden probar ejecutándola directamente.)

b) Realizar un programa similar a la función del apartado (a) pero que pueda ser ejecutado desde la línea de comando y muestre el resultado. Es decir, se puede ejecutar así:

```
C:\>  calcula + 5 6 4 10
```

y muestre el 25. Por supuesto la lista de números será indeterminada.

Ejercicio 5(Opcional)

Un juego con elementos móviles en consola (tipo space invaders sencillo por ejemplo). Sería similar al planteado en el primer tema pero ya con más posibilidades y usando los elementos vistos en este tema.

Ejercicio 6 (Opcional)

Mastermind: El ordenador saca una combinación de cuatro símbolos (o colores) de siete posibles y el usuario tiene que adivinarlo. En cada turno el usuario da una combinación y el ordenador le dice cuantos aciertos de símbolo hay y cuantos aciertos de símbolo y posición hay. Usa vectores de 4 elementos. Se deben guardar todas las tiradas del usuario en una colección para hacer un informe final de número de tiradas, momento de mayor acierto (salvo el final si gana el usuario), momento de meno aciertoy otras que se te ocurran. Se debe poder abandonar el juego en la mitad que significaría que pierde el usuario. Puedes intentar hacerlo en modo gráfico.

Más información en: http://es.wikipedia.org/wiki/Mastermind

Apéndice I: Número indeterminado de parámetros

Mediante la palabra reservada **params** se puede hacer que un método acepte un array de parámetros opcionales.

Siempre se pasa por valor y será la última definición.

```
tipo_dato nombre_funcion(parametros fijos, params tipoarray[]
nombrearray)
```

Como es una matriz, solo se puede disponer de un tipo de dato dentro de dicho vector, si embargo podemos simular varios tipos de dato mediante sobrecarga.

```
public static int suma(params int[] parametros)
 {
     int i;
     int a = 0;

     for (i = 0; i < parametros.Length; i++)
         a += parametros[i];
     return a;
 }

 public static double suma(params double[] parametros)
 {
     int i;
     double a = 0;

     for (i = 0; i < parametros.Length; i++)
         a += parametros[i];
     return a;
 }
```

Llamada:

```
Console.WriteLine(suma(2, 3, 3, 4, 5, 3, 77, 6, 7));
Console.WriteLine();
Console.WriteLine("{0,0:.00}", suma(2, 4.44, 3, 4.76, 3.45, 4.56, 5.6, 7));
```

También se puede aprovechar el polimorfismo mediante Object para pasar distintos tipos en la misma función:

```
public static void funcion(params object[] parametros)
```

Apéndice II: El método Main

Hasta ahora sólo se ha visto una versión de *Main*() que no toma parámetros y tiene como tipo de retorno *void*, pero en realidad todas sus posibles versiones son:

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

Estas variantes sirven para conectar la aplicación con el "exterior", es decir, con el Sistema Operativo.

Devolución:

Hay versiones de *Main*() que devuelven un valor de tipo *int.* Dicho número sería interpretado como código de retorno de la aplicación. Éste valor suele usarse para indicar si la aplicación a terminado con éxito (generalmente valor 0) o no (valor según la causa de la terminación anormal). Este puede ser usado por el Sistema Operativo o por otra aplicación que llama a esta para realizar alguna operación.

Veamos un ejemplo para entender esto. Vamos a ejecutar el Notepad desde nuestra aplicación de C# y veremos su código de retorno. Para ello escribir el siguiente código:

```
Process p;                          // Creamos un objeto proceso
p = Process.Start("Notepad");       // Ejecuta un nuevo programa
p.WaitForExit();                    // Espera a que el programa termine
Console.WriteLine(p.ExitCode);      // Muestra código de retorno(Exitcode)
```

Si todo ha ido correctamente, esto mostrará en pantalla un 0. Este código puede ser usado para tomar alguna decisión si se ha ejecutado un programa y ha salido algo mal

Lista de argumentos:

También hay versiones de *Main*() que toman un parámetro donde se almacenará la lista de argumentos con los que se llamó a la aplicación, por lo que sólo es útil usar estas versiones del punto de entrada si la aplicación va a utilizar dichos

argumentos para algo. El tipo de este parámetro es *string*[], lo que significa que es una tabla de cadenas de texto, y su nombre -que es el que habrá de usarse dentro del código de *Main*() para hacerle referencia- es *args* en el ejemplo, aunque podría dársele cualquier otro.

Veamos un ejemplo:

```
using System;

class Formatea
{
    public static int Main(string[] args)
    {
      int i;

      if (args.Length == 0)
      {
          Console.WriteLine("No ha pasado ningún argumento. Se debe usar:");
          Console.WriteLine("\t-M: Paso a mayusculas\n\t-m: Paso a minusculas");
          Console.WriteLine("y a continuacion la cadena a formatear");
          return 2; // Codigo de "no argumentos"
      }
      else
          switch (args[0]) {
            case "-m":
                  for (i=1; i<args.Length; i++)
                        Console.Write(args[i].ToLower()+' ');
                  Console.WriteLine ();
                  break;
            case "-M":
                  for (i=1; i<args.Length; i++)
                        Console.Write(args[i].ToUpper()+' ');
                  Console.WriteLine ();
                  break;
            default:
                  Console.WriteLine("Argumento no valido. Se debe usar:");
                  Console.WriteLine("\t-M: Paso a mayusculas");
                  Console.WriteLine("\t-m: Paso a minusculas");
                  return 1; // Codigo de argumento no valido
             }
      return 0; //Codigo de "todo correcto"
    }
}
```

Apéndice III: Clases indexadas

Es posible en C# extender la indización de un elemento interno al objeto en general. Esto se realiza mediante set/get de la propiedad this[int] como se ve en el ejemplo:

```
class Prueba
 {
     private int[] v = new int[100];

     public int this[int indice]
     {
         set
         {
             v[indice] = value;
         }
         get
         {
             return v[indice];
         }
     }

     // Otros métodos y propiedades de la clase Prueba

 }
 class Program
 {
     static void Main()
     {
         Prueba p = new Prueba();
         p[3] = 12;
         Console.WriteLine(p[3]);
         Console.ReadKey();
     }
 }
```

Apéndice IV: Estructuras

Una estructura es lo que se llama un "objeto ligero" y de hecho se usa y define prácticamente igual que un objeto.

Al igual que una clase, una estructura puede tener tanto variables como métodos, pero **no admiten herencia** (aunque sí pueden implementar interfaces). Además de este punto, la gran diferencia con las clases es que las estructuras son variables tipo valor y **no es necesario instanciarlas** como las clases. En definitiva, no son referencias.

Se utilizan cuando es necesario manejar objetos ligeros (de pocas propiedades) de forma eficiente, ya que el acceso a sus miembros es más rápida (no así la asignación de estructuras que equivale a hacer una copia entera de la estructura).

Se define igual que una clase pero usando la palabra reservada *struct* en lugar de *class.* Por ejemplo:

```
struct Colega
{
    public string Nombre;
    public DateTime FechaNac;
    public string email;
}
```

En el Main():

```
        Colega colega1;
        Colega colega2, colega3;
        colega1.Nombre = "J.";

        colega1.FechaNac = new DateTime(1980, 10, 23);
        colega1.email = "jota@supermail.com";

        colega2.Nombre = "La Juana";
        colega2.FechaNac = new DateTime(1972, 9, 29);
        colega2.email = "juanilla@granmail.com";

        colega3 = colega1;
        colega3.Nombre = "J Power";

        Console.WriteLine(colega1.Nombre);
```

Se puede realizar un **new** de una estructura con el objetivo de llamar a un posible constructor definido o para inicializar sus campos con valores por defecto.

Ten en cuenta que también son objects por herencia aunque luego no se pueda heredar.