UD 2 ASP.NET Core. Características. El lenguaje C#.
2.4 Arrays y listas.

En esta parte veremos cómo trabajar con arrays y listas en C#.

Se recomienda utilizar como referencia el capítulo 2 del siguiente libro:

https://www.syncfusion.com/ebooks/csharp

Además de los siguientes apartados de la guía de programación de C#:

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/arrays/index

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/collections

Arrays.

Hasta ahora hemos trabajado sólo con tipos de datos que permiten almacenar un único elemento, pero es habitual que necesitemos resolver tareas para las que tengamos que trabajar con conjuntos de datos.

Para declarar una variable de tipo array o matriz en C# utilizamos corchetes, del mismo modo que en java:

```
tipo[] nombreArray;
```

En C#, las matrices son objetos, y no simplemente regiones direccionables de memoria contigua como en C y C++. Array es el tipo base abstracto de todos los tipos de matriz y podemos usar las propiedades y métodos que nos proporciona. Un ejemplo típico es el uso de la propiedad Length para obtener la longitud de una matriz, como en el ejemplo:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

Inicialización de arrays.

Para inicializar una matriz con sus valores predeterminados usamos el operador *new*, y pasando entre corchetes el número de elementos que almacenará el array, como en el siguiente código:

```
string[] stringArray = new string[6];
```

Es posible inicializar una matriz especificando sus valores iniciales, en cuyo caso, el especificador de rango no es necesario porque ya lo proporciona el número de elementos de la lista de inicialización:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

Tenemos también un método abreviado para inicializar la matriz con valores iniciales, sin utilizar el operador *new*. Este método abreviado sólo se puede utilizar cuando declaramos e inicializamos el array en la misma sentencia, si se declara y es inicializada más adelante tenemos que utilizar siempre el operador *new*:

```
int[] array2 = { 1, 3, 5, 7, 9 };
```

Matrices multidimensionales.

Hasta ahora hemos visto sólo arrays unidimensionales, pero podemos declarar arrays de varias dimensiones, usando la sintaxis que podemos ver a continuación:

```
int[,] array = new int[4, 2];
int[, ,] array1 = new int[4, 2, 3];
```

En estos dos ejemplos tenemos una matriz bidimensional, con cuatro filas y dos columnas y una matriz de tres dimensiones, 4, 2 y 3.

Vemos a continuación cómo inicializar la primera matriz con la notación normal y la abreviada:

```
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 }};
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Como Podemos ver, como la primera dimensión es 4, especificamos 4 conjuntos de valores entre corchetes, con 2 elementos en cada grupo, como corresponde a la segunda dimensión que hemos declarado, separados por comas. Estos conjuntos van a su vez entre corchetes y separados entre sí por comas, completando la declaración del array.

Recorrer matrices.

Una tarea que nos encontramos muy a menudo consiste en recorrer todos los elementos de un array, por ejemplo para mostrarlos al usuario o para buscar un valor que cumpla determinada condición dentro de los elementos del array.

Aunque para recorrer un array podemos utilizar un bucle *for* convencional, existe una construcción más sencilla, que fue diseñada precisamente para recorrer los elementos de una colección, y es *foreach*.

A continuación se muestra un ejemplo en el que se muestran los elementos de un array empleando una construcción *foreach*:

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Muestra en pantalla: 4 5 6 1 2 3 -2 -1 0
```

Como podemos ver, en la sentencia *foreach* se especifica el tipo de dato que contiene la colección, *int* en nuestro caso. A continuación se asigna un nombre que será el de la variable que contenga los diversos valores de la colección, según el bucle los vaya recorriendo. En último lugar se especifica la colección que queremos recorrer o iterar, utilizando el operador *in* seguido del nombre de la variable que contiene la colección de datos.

Propiedades y métodos útiles para el trabajo con arrays.

Como hemos comentado anteriormente, cualquier array que declaremos hereda de la clase base Array, que proporciona una gran cantidad de métodos y propiedades que podemos utilizar al trabajar con matrices.

A continuación haremos mención a algunos de ellos pero en el siguiente enlace se puede consultar la documentación de la clase Array para ver todos los disponibles:

https://docs.microsoft.com/es-es/dotnet/api/system.array?view=netcore-2.1

- ***Length***

Devuelve el número de elementos del array y es una de las propiedades más útiles. El uso más común es para conocer el límite superior cuando recorremos la matriz con un bucle *for.*

```
nombreArray.Length
```

- ***Sort***

Es un método estático de la clase Array, que permite ordenar los valores del array.

```
Array.Sort()
```

- ***Reverse***

Método estático de la clase Array, que permite invertir el orden de los valores del array.

```
Array.Reverse()
```

- ***Find***

Método estático de la clase Array, que permite buscar un valor que cumpla una determinada condición dentro de un array.

```
Array.Find()
```

Strings como arrays.

Una cadena de texto puede ser tratada también como una matriz unidimensional, donde cada elemento de la misma sería un carácter. De este modo podríamos recorrer la misma con un bucle, para buscar un determinado elemento dentro de la cadena o realizar otro tipo de operaciones relacionadas con los arrays.

En el siguiente enlace se puede consultar la documentación de la clase String, con todas las propiedades y métodos que ofrece, pero a continuación mencionaremos algunos que pueden resultar interesantes:

https://docs.microsoft.com/es-es/dotnet/api/system.string?view=netcore-2.1

- ***Length***

Al igual que en cualquier array devuelve el número de elementos. En este caso se tratará del número de caracteres de la cadena de texto.

```
nombreString.Length
```

- ***IndexOf***

Devuelve la posición de la primera aparición de un carácter en la cadena, o -1 si no se encuentra dicho carácter.

```
nombreString.IndexOf()
```

- ***ToUpper / ToLower***

Permite obtener la cadena de texto, convertidos todos los caracteres a mayúsculas o a minúsculas.

```
Array.Reverse()
```

- ***Split***

Este es un método muy potente, que permite dividir una cadena de texto en una serie de subcadenas especificando el carácter por el que serán divididas. Por ejemplo, podemos obtener todas las palabras de una cadena de texto dividiendo las mismas especificando como separador el carácter espacio. El resultado es un array que contiene las cadenas en las que se ha dividido.

```
Array.Split()
```

Colecciones.

Hemos visto uno de los modos de trabajar con agrupaciones de objetos: mediante la creación de matrices de objetos. Las matrices son muy útiles cunado trabajamos con un número **fijo** de objetos.

Hay ocasiones en las que, sin embargo, no conocemos el número de objetos con los que vamos a trabajar. Las colecciones proporcionan una manera más flexible de trabajar con grupos de objetos. A diferencia de las matrices, el grupo de objetos con el que trabaja puede aumentar y reducirse de manera dinámica a medida que cambian las necesidades de la aplicación.

En este enlace encontramos la documentación sobre colecciones en C#:

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/collections

Existen diversos tipos de colecciones en C#, como pueden ser los diccionarios, colas, pilas etc.. Todos ellos están diseñados para ser utilizados en escenarios concretos y se puede consultar la documentación en el enlace anterior. A continuación vamos a ver brevemente las listas, que serán las colecciones que utilicemos más a menudo.

Listas en C#. List<T>.

El lenguaje C# nos proporciona la clase genérica *List<T>* para crear colecciones de objetos del tipo de dato especificado como *T*. Esto permite crear una colección de objetos fuertemente tipados, con la ventaja de que la colección puede aumentar o disminuir de tamaño de forma dinámica, añadiendo o eliminando objetos de la lista.

En primer lugar vemos un ejemplo donde se crea una lista y se añaden elementos de forma dinámica con el método *Add*:

```
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");
```

Cuando inicializamos la lista con el operador *new*, no contiene ningún elemento inicialmente, pero con cada nuevo objeto que añadimos con *Add* el tamaño de la colección crece.

Si el contenido de una colección se conoce de antemano, puede usar un inicializador de colección para inicializar la colección. A continuación se muestra el ejemplo anterior, utilizando un inicializador de colección para agregar elementos a la colección en el momento de la declaración:

```
var salmons = new List<string> {"chinook", "coho", "pink", "sockeye"};
```

Al igual que ocurre con las matrices, podemos recorrer nuestra colección con un bucle *for* convencional, accediendo a cada elemento de la colección mediante su índice o posición en la misma o con un bucle *foreach*. Los dos siguientes fragmentos de código serían equivalentes:

```
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
```

```
for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
```

Se puede observar una pequeña diferencia con las matrices, ya que en aquellas utilizábamos la propiedad *Length* para obtener el número de elementos, mientras que en una colección obtenemos la misma información con la propiedad *Count*. Esto tiene sentido debido a que la longitud de un array es fija, y la propiedad devuelve ese valor, en una colección no tenemos una longitud fija.

Hay un caso concreto en el que no es posible recorrer una colección utilizando *foreach*, y es cuando recorremos una colección, eliminando elementos que cumplan determinada condición. Esto es así porque el método *RemoveAt* hace que el elemento en una posición concreta sea eliminado de la lista, desplazándose a la izquierda todos los demás elementos de la misma. Por esta razón, para esta tarea recorreremos la lista con un bucle *for*, que permite recorrer la lista en orden inverso. De esta forma al eliminar un elemento, no importa que se recoloquen los elementos a continuación, ya que en el siguiente paso del bucle pasaremos al objeto previo, no al posterior. Lo vemos en el siguiente ejemplo:

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Vamos eliminando un objeto si y otro no
        numbers.RemoveAt(index);
    }
}
```

Ejemplo.

Vamos a volver al ejercicio de la parte anterior, sobre métodos, donde se pide una página que muestre todos los números primos entre uno y un número pasado como parámetro.

En aquella ocasión se decidió todo el código en la vista, teniendo un bucle que recorría los números y tras determinar si eran primos utilizando un método estático, se mostraban o no en la página.

Aunque funciona correctamente, no es la manera más recomendable de realizarlo, ya que en la vista no deben realizarse ese tipo de cálculos, dejando únicamente el código necesario para mostrar y formatear los datos de salida. Como ya somos capaces de almacenar una colección completa de valores, veremos la forma adecuada de resolver el ejercicio.

Comenzamos por el controlador:

HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;

namespace Ejercicio.Controllers
{
    public class HomeController : Controller
    {
        private static int count = 0;

        public IActionResult Index(int number)
        {
            List<int> numbers = new List<int>();

            for (int i = 1; i <= number; i++)
            {
                if(IsPrime(i))
                {
                    numbers.Add(i);
                }
            }

            ViewData["n"] = number;
            ViewData["numbers"] = numbers.ToArray();

            return View();
        }

        private bool IsPrime(int n)
        {
            for (int i = 2; i <= Math.Sqrt(n); i++)
            {
                if (n % i == 0)
                {
                    return false;
                }
            }

            return true;
        }
```

```
    }
}
```

Ya no necesitamos que el método *IsPrime* sea estático, ya que sólo será llamado desde dentro de la propia clase.

Comenzamos declarando una lista de enteros, ya que sabemos que todos los elementos que vamos a almacenar son números enteros pero no sabemos cuantos números primos vamos a encontrar entre 1 y el parámetro que obtendremos de la petición url, que es *number*.

A continuación, recorremos todos los números y comprobamos si son primos utilizando nuestra función. Si es así los añadimos a la colección de primos encontrados.

Finalmente guardamos en el *ViewData* el valor de *number*, que será utilizado en la vista para el título y, ahora sí, vemos que podemos guardar en *ViewData* toda una colección de números pasando una colección o un array. En este caso podemos observar que hemos utilizado el método *ToArray()*, que devuelve un array que contiene una copia de todos los elementos de una colección, en el mismo orden. Esto no sería necesario y podríamos pasar a la vista la colección directamente. La razón de hacerlo es que los arrays son más eficientes y rápidos que las colecciones, y como en la vista no vamos a añadir o quitar elementos, si no que vamos simplemente a recorrer y mostrar los elementos, un array es perfecto para esta tarea.

La vista quedaría del siguiente modo:

Index.cshtml

```
<html>
<head>
    <title>Ejercicio 2.3.4</title>
</head>
<body>
    <h2>Números primos entre 1 y @ViewData["n"]</h2>
    <ul>
        @foreach (var item in (int[])@ViewData["numbers"])
        {
                <li>@item</li>
        }
    </ul>
</body>
</html>
```

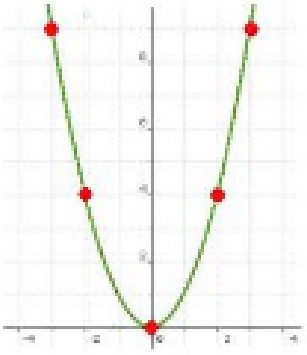
En este caso, en lugar de tener un bucle de 1 a *number* y comprobar si los números son primos, que es una tarea más propia del controlador, nos encontramos con una colección de valores que ya ha sido calculada y que simplemente debemos mostrar. Ese es el tipo de tarea del que se debe encargar la vista.

Como *ViewData* es un contenedor de datos de cualquier tipo, debemos hacer un *cast* o conversión a *int[]* para indicar al compilador que el dato etiquetado como “numbers” y que queremos recorrer es un array de enteros. A continuación lo recorremos con un *foreach* y simplemente mostramos cada elemento recorrido en nuestra lista.

De este modo el código de la vista es mucho más sencillo y limpio y lo más importante, separamos las tareas de modo que cada elemento se encarga de su cometido.

Ejercicio 2.4.1.

Queremos una aplicación web que permita resolver ecuaciones cuadráticas. Para ello indicaremos como parámetros los valores de los coeficientes a, b y c (números enteros).



$$ax^2 + bx + c = 0$$

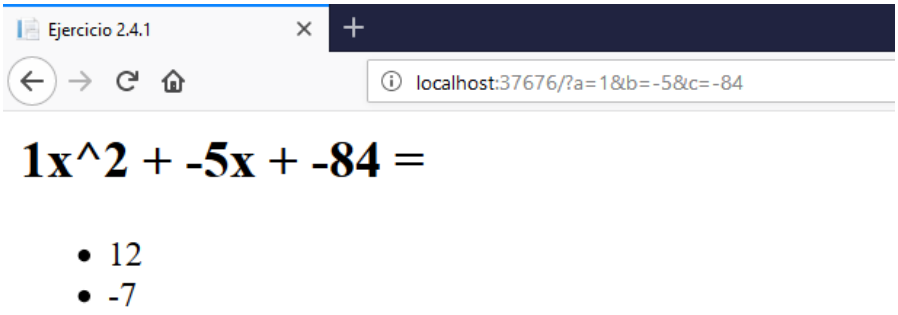
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Para el cálculo de la solución tendremos en cuenta los siguientes casos:

a. Si a==0 y b==0, mostrará un mensaje diciendo que la ecuación es degenerada.
b. Si a==0 y b!=0, se mostrará el único resultado cuyo valor es -c/b.
c. En el resto de los casos, tenemos 2 soluciones que se mostrarán en la página de salida.

Pista: Una posible solución podría ser que el controlador devuelva los resultados a la vista en una colección. En la vista se podría comprobar el número de resultados que contiene la solución, de forma que ya sabríamos si tenemos dos, una o ninguna solución a la ecuación.

A continuación se muestra un ejemplo de ejecución:

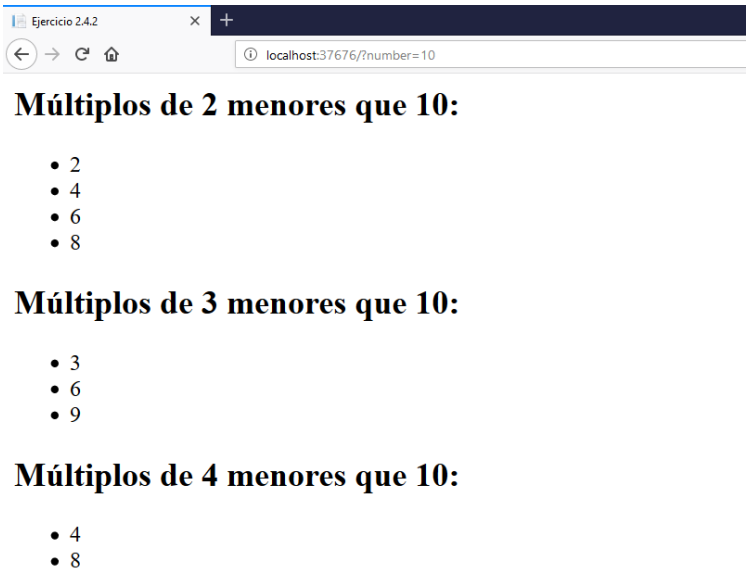


Ejercicio 2.4.2.

En este caso queremos una aplicación que a partir de un número indicado por el usuario:

1) Invoque a una función para que calcule todos os números pares (múltiplos de 2) menores que el número indicado.
2) Invoque a la misma función, para que esta vez calcule todos los múltiplos de 3 menores que el número indicado.
3) Invoque de nuevo la misma función, para calcular los múltiplos de 4 menores que el número indicado.

Por último, todos estos números serán mostrados en una página en el navegador.

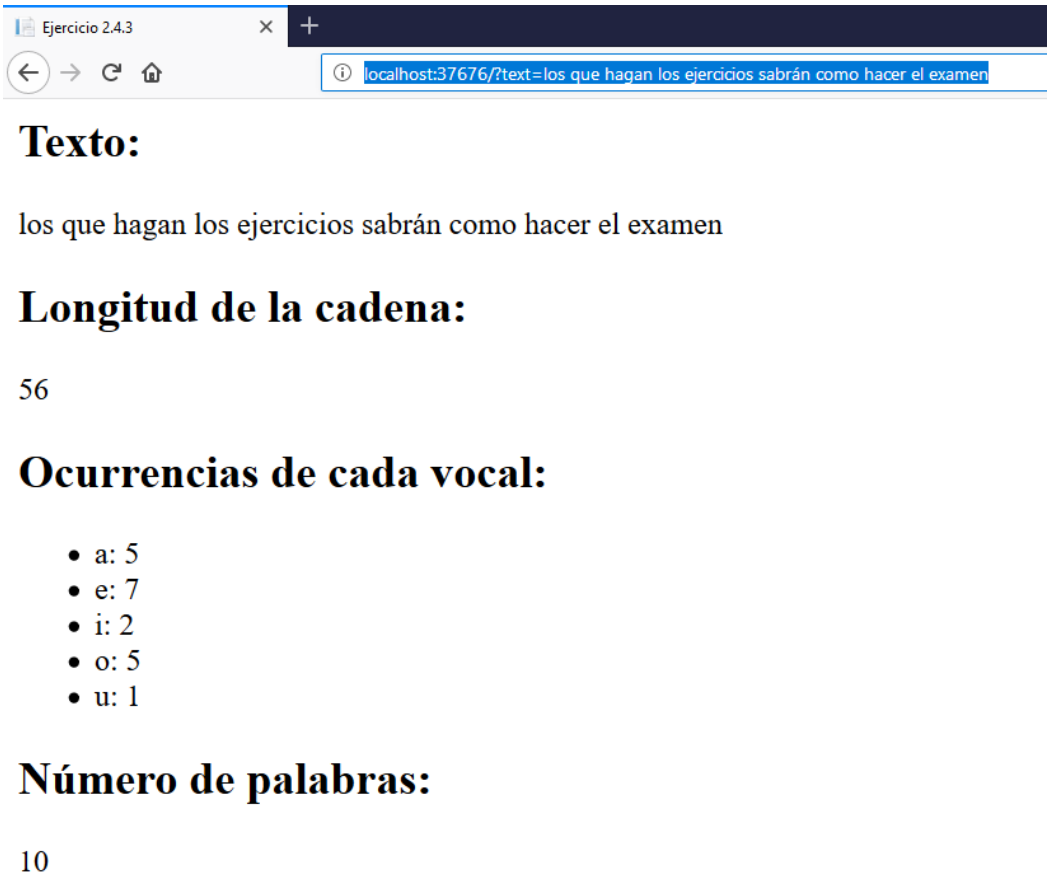


Ejercicio 2.4.3.

En este ejercicio realizaremos una serie de tareas con una cadena de caracteres. Para ello el usuario introducirá una cadena de texto como parámetro en la url.

A continuación la página de salida debe mostrar una serie de datos sobre la cadena. Se enumera la información que se debe mostrar:

- La propia cadena de texto.
- Número de caracteres de la cadena o longitud.
- Número de ocurrencias de cada vocal (sin distinguir mayúsculas de minúsculas) en la cadena.
- Número de palabras de la cadena (se considera que las palabras están separadas por un espacio en blanco).



Ejercicio 2.4.4.

Para la realización de este último ejercicio esperaremos también una cadena como parámetro. En nuestra página se mostrará el resultado que se obtiene cada vez que se realiza una rotación de un carácter a la derecha sobre dicha cadena, hasta que se obtenga de nuevo la cadena original.