

UD 2 ASP.NET Core. Características. El lenguaje C#.

2.2 Estructuras de control.

En esta segunda parte de la unidad se presentarán las instrucciones de control, que permiten modificar el flujo secuencial de ejecución del código.

Al igual que en la primera parte podemos encontrar información de referencia en el capítulo 2 del siguiente libro:

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

Como siempre, también podemos utilizar la documentación oficial del lenguaje:

<https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/statements-expressions-operators/statements>

Instrucciones de selección.

Una instrucción de selección hace que el control del programa se transfiera a un determinado flujo dependiendo de si cierta condición es cierta o no:

<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/selection-statements>

A continuación se presentan las instrucciones comunes, que como se puede observar son similares a las de Java y otros lenguajes basados en la sintaxis de C/C++.

if

La instrucción *if* es una de las más importantes de cualquier lenguaje de programación, y es la base de la selección del flujo de ejecución basada en una condición.

La sintaxis es la siguiente:

```
if (condition)
{
    sentence;
}
```

La condición puede ser cualquier expresión que pueda ser evaluada con un valor *bool*, es decir, un resultado *true* o *false*. Si la expresión se evalúa a *true* se ejecutaría el código contenido en el bloque entre llaves, ignorándose en caso contrario.

Existe la posibilidad de utilizar una sintaxis abreviada sin utilizar un bloque entre llaves siempre y cuando el código a ejecutar sea sólo una única línea, tal y como se indica a continuación:

```
if (condition)
    sentence;
```

Aunque esta notación es muy habitual y es común ver código escrito de esta forma, la recomendación es utilizar siempre un bloque entre llaves, ya que favorece la legibilidad del código y las posibles modificaciones posteriores del mismo.

C# permite anidar tantas instrucciones *if* dentro de otras como necesitemos.

else

En ocasiones la simple ejecución de un código cuando una condición es cierta no es suficiente y lo que deseamos es establecer dos vías de ejecución de modo que si la expresión es cierta sigamos una de estas vías o la otra en caso contrario. Para ello disponemos de la instrucción *else*, que permite establecer una alternativa para cuando la condición no sea evaluada a *true*.

```
if (condition)
{
    sentenceA;
}
else
{
    sentenceB;
}
```

else if

Por último, también es posible anidar una nueva sentencia de evaluación *if* dentro de una sentencia *else*, lo que permite una gran flexibilidad sobre el flujo del código.

```
if (conditionA)
{
    sentenceA;
}
else if (conditionA)
{
    sentenceB;
}
else
{
    sentenceC;
}
```

ejemplo

El ejemplo siguiente determina si un carácter de entrada es una letra minúscula, una letra mayúscula o un número. Si estas tres condiciones son false, el carácter no es un carácter alfanumérico. En el ejemplo se muestra un mensaje para cada caso.

```
Console.Write("Enter a character: ");

char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
```

```
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}
```

switch

Si en función del valor de una expresión necesitamos ejecutar fragmentos de código diferentes, a veces resulta engorroso utilizar sentencias *if-else* anidadas. Para facilitar estas situaciones podemos utilizar la sentencia *switch*, que es capaz de seleccionar un código a ejecutar en función de la comparación del resultado de una expresión con una serie de patrones de coincidencia.

Ejemplo:

```
public enum Color { Red, Green, Blue }

Color c = (Color)(new Random()).Next(0, 3);
switch (c)
{
    case Color.Red:
        Console.WriteLine("The color is red");
        break;
    case Color.Green:
        Console.WriteLine("The color is green");
        break;
    case Color.Blue:
        Console.WriteLine("The color is blue");
        break;
    default:
        Console.WriteLine("The color is unknown.");
        break;
}
```

El funcionamiento es el siguiente. En primer lugar se evalúa la expresión de la sentencia *switch*. A continuación el valor de esta expresión se va comparando con una serie de valores especificados en tantos *case* como opciones nos interesan. En caso de que esa comparación sea cierta, se ejecuta el código que tengamos dentro de ese caso, hasta llegar a una sentencia *break* en caso de que exista. En caso afirmativo terminaría la ejecución del *switch*, pero en caso contrario seguirían evaluándose los siguientes casos. Podemos definir un caso por defecto *default*, que sería ejecutado si ninguno de los casos coincide con nuestra expresión o si aún existiendo una coincidencia no terminamos la misma con una sentencia *break*.

Instrucciones de iteración.

Es posible crear bucles de ejecución mediante las instrucciones de iteración. Las instrucciones de iteración producen secuencias de sentencias que se ejecutarán varias veces, según los criterios de la finalización de bucle. Estas instrucciones se ejecutan en orden, excepto cuando se detecta una instrucción de salto.

<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/iteration-statements>

while

La instrucción *while* ejecuta una instrucción o un bloque de instrucciones mientras una expresión booleana especificada se evalúa como true. Como esa expresión se evalúa antes de cada ejecución del bucle, un bucle *while* se ejecuta cero o varias veces.

```
while (condition)
{
    Sentences;
}
```

do while

La instrucción *do* ejecuta una instrucción o un bloque de instrucciones mientras una expresión booleana especificada se evalúa como true. Como esa expresión se evalúa después de cada ejecución del bucle, un bucle *do-while* se ejecuta una o varias veces.

```
do
{
    Sentences;
} while (condition)
```

for

La instrucción *for* ejecuta una instrucción o un bloque de instrucciones mientras una expresión booleana especificada se evalúa como true. Permite también especificar una sentencia de inicialización y una sentencia de iteración que hará que un valor vaya cambiando hasta que en un momento determinado la condición deje de ser cierta.

```
for (initializer; condition; iterator)
{
    Sentences;
}
```

foreach

La instrucción *foreach* es una construcción que facilita recorrer una colección de elementos.

```
foreach (var element in collection)
{
    Sentences;
}
```

Instrucciones de salto.

La creación de una rama se realiza mediante instrucciones de salto, que producen una transferencia inmediata del control del programa. Las siguientes palabras clave se usan en instrucciones de salto.

<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/jump-statements>

break

La instrucción *break* finaliza la ejecución del bucle o de la instrucción *switch* en la que aparezca. El control se pasa a la instrucción que haya a continuación de la instrucción finalizada, si existe.

continue

La instrucción *continue* transfiere el control a la siguiente iteración de la instrucción envolvente *while*, *do*, *for* o *foreach* en la que aparece.

return

La instrucción *return* termina la ejecución del método en el que aparece y devuelve el control al método de llamada. También puede devolver un valor opcional. Si el método es del tipo *void*, la instrucción *return* se puede omitir.

Ejemplo.

Vamos a ver ahora un pequeño ejemplo en el que apliquemos lo aprendido, en este caso utilizando una sencilla sentencia de iteración o bucle.

Lo que vamos a hacer es que nuestra aplicación web muestre una página en el navegador en la que se muestre una tabla html que contenga 6 filas, de modo que en cada una de ellas se indique en primer lugar el número de fila y a continuación el mensaje "DWES - Ejemplo de bucle".

En primer lugar vamos a analizar la tarea a realizar. Si nos fijamos bien, en este caso no necesitamos realmente que nuestra aplicación realice ningún cálculo, siendo la tarea que se nos pide eminentemente relacionada con la presentación de datos al usuario. Por esta razón, nos encontramos con un caso en el que si está recomendado insertar la mayor parte del código en la vista, y no en el controlador. Esto se debe a que estamos ante una tarea de formateo de los datos de salida, y ese tipo de tareas se realizan en la vista, es este caso concreto debemos generar una tabla de forma dinámica. A continuación veremos como.

Empezamos por nuestro controlador, que en esta ocasión es muy sencillo y se limita a definir un mensaje que es el que pasaremos a la vista para que lo muestre repetido. También vamos a definir el número de veces que vamos a repetir el mensaje. El definir tanto el mensaje como el número de veces como variables y pasarlas a la vista hace el código mucho más flexible ya que si quiero cambiar el mensaje o el número de filas de la tabla, bastaría con cambiar el valor de dichas variables.

```
public IActionResult Index()
{
    ViewData["rows"] = 6;
    ViewData["message"] = "DWES - Ejemplo de bucle";

    return View();
}
```

Como decíamos, la mayor parte del trabajo tendrá lugar en la vista, que se muestra a continuación:

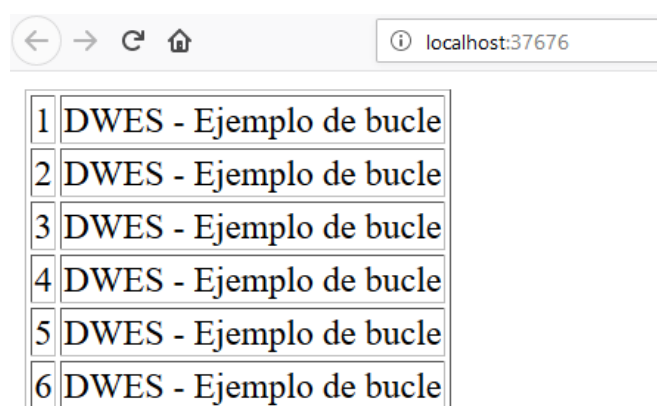
```
<html>
<head>
    <title>Ejemplo unidad 2.2</title>
</head>
<body>
    <table border="1" >
        @for(int line = 1; line <= (int)ViewData["rows"]; line++)
        {
            <tr>
                <td>@line</td>
                <td>@ViewData["message"]</td>
            </tr>
        }
    </table>
</body>
</html>
```

Vamos a pararnos a analizar el código para comprender que es lo que hemos hecho. Lo primero que nos puede llamar la atención es el modo en el que estamos mezclando código C# con el html de la página, vamos a intentar aclararlo.

En primer lugar, el enunciado nos pedía mostrar los resultados en una tabla, por lo que insertamos una en nuestro html. Como queremos mostrar un número de filas indicado en nuestro código, empleamos para ello un bucle *for*. El bucle utiliza una variable que nos servirá además de como contador del bucle, para mostrar el número de línea en la tabla. El número de filas se ha pasado en la ViewData desde el controlador. Como el ViewData puede almacenar cualquier tipo de valor, el compilador no puede saber si se trata de un entero o de cualquier otra cosa, por lo tanto hacemos un cast para forzar a que el valor contenido en el ViewData sea tratado como un *int*. En este momento tenemos un bucle que se va a repetir tantas veces como queremos.

A continuación queremos mostrar filas en la tabla como veces se ejecute el bucle, el problema es que dentro del bloque de código del bucle debemos especificar elementos de html. El procesador de páginas de ASP.NET Core hace esto muy sencillo, ya que lo que hace es trabajar en dos “modos”, modo código y modo html. En nuestro caso como estamos dentro del bloque de un bucle estamos trabajando con código, pero simplemente con introducir una etiqueta html hacemos que el procesador se de cuenta y pase a “modo html”. De esta forma empieza a escribir el código html en la salida, creando en primer lugar una fila <tr>. Para el contenido de la primera celda necesitamos insertar de nuevo código, y el procesador es capaz de darse cuenta porque encuentra un símbolo @ seguido de una variable. Para la segunda celda inserta el contenido del mensaje pasado al ViewData. Por último, cuando llega a la llave de cierre, se da cuenta de que hemos vuelto al código, saltando a la siguiente ejecución del bucle. De este modo podemos integrar nuestro código con el html para generar nuestra tabla de forma dinámica.

Si ejecutamos nuestra aplicación encontraremos algo como esto:



1	DWES - Ejemplo de bucle
2	DWES - Ejemplo de bucle
3	DWES - Ejemplo de bucle
4	DWES - Ejemplo de bucle
5	DWES - Ejemplo de bucle
6	DWES - Ejemplo de bucle

Ejercicio 2.2.1.

En este primer ejercicio utilizaremos un bucle para realizar un cálculo acumulativo.

Queremos crear una aplicación web que muestre en una página el resultado de ir sumando consecutivamente los números desde 1 a 50, es decir $1 + 2 + 3 + 4 + \dots + 50 = 1275$.

El resultado debe ser similar al siguiente:

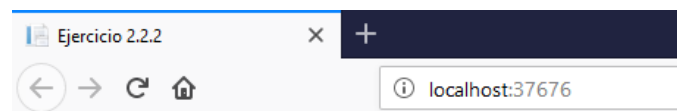


La suma acumulada de los números de 1 a 50 es: 1275

Los valores del resultado deben ser calculados en el controlador y pasados a la vista.

Ejercicio 2.2.2.

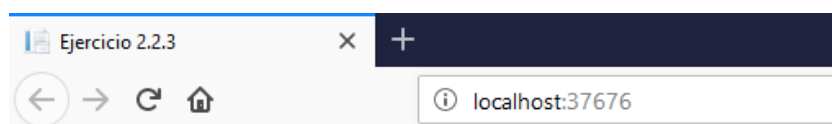
En este ejercicio queremos que nuestra aplicación muestre una página en el navegador donde se muestre la tabla de multiplicar del 1.



1 x 1 = 1
2 x 1 = 2
3 x 1 = 3
4 x 1 = 4
5 x 1 = 5
6 x 1 = 6
7 x 1 = 7
8 x 1 = 8
9 x 1 = 9
10 x 1 = 10

Ejercicio 2.2.3.

En este caso lo que queremos es una página donde se muestren todos los números de 1 a 100 que sean múltiplos de 2 o de 3.



Múltiplos de 2 y 3 entre 1 y 100

- 2
- 3
- 4
- 6
- 8
- 9
- 10
- 12
- 14
- 15
- 16
- 18
- 20
- 21
- 22
- 24
- 26
- 27