Bases de la programación ($C\#$)

Carlos Clement Bellido

18 de junio de 2019

Índice general

Prólogo

Atención: este documento ha sido escrito como forma de apoyo para dar clases, puede que algunas cosas no queden del todo claras o no tengan una definición precisa (intentaré ser lo más claro y conciso posible). No dudes en contactar conmigo para sugerencias, correcciones o lo que se pase por tu mente.

carlos.clement.bellido@gmail.com

Algún rollo sobre la programación en inglés
Si es tu primer contacto con la programación, tómatelo con paciencia y no te rindas, lo que se te viene encima no es sencillo. *Si vis pacem, para bellum.*

Capítulo 1

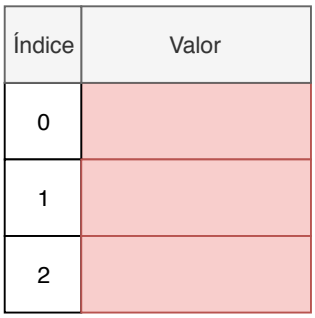
Tipos enumerables

1.1. Matriz (*array*)

Las matrices, o *arrays*[1], son agrupaciones de un determinado número elementos del mismo tipo. La cuantía ha de definirse en la inicialización de la matriz y, por lo general, no se alterará. Tenemos tres tipos básicos: la unidimensional, que es la más común de ver, la multidimensional, la cual nos permitirá trabajar con varias dimensiones, y la escalonada, que es similar a la multidimensional con la salvaguarda de que no se requiere que todas las dimensiones tengan el mismo tamaño.
Un concepto muy importante a tener en cuenta es que tenemos que comprender que **una matriz es un tipo de datos** también.

1.1.1. Unidimensional

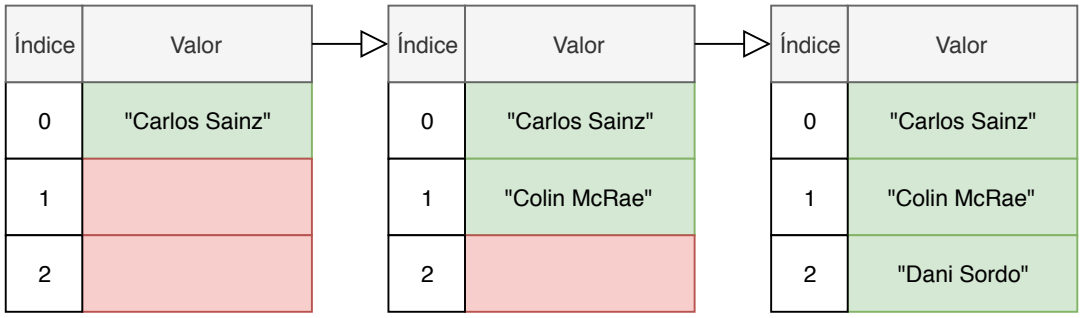
En una matriz unidimensional[2] cada dato le corresponde a un índice, que será un número. De este modo, podemos añadir datos a la matriz (siempre y cuando haya espacios libres).
Pongamos el ejemplo de que queremos guardar los nombres de los tres primeros pilotos que llegan a meta. Sabemos que siempre tendremos que guardar tres nombres, así que de primeras inicializamos la matriz:



Los valores que guardemos serán de un tipo determinado; como en este caso guardaremos nombre de pilotos, el tipo de datos sera un tipo cadena (*string*). Los tres pilotos que queremos meter serán:

1. Carlos Sainz
2. Colin McRae
3. Dani Sordo

Así que, uno a uno, los insertaremos hasta que rellenemos la matriz.



De izquierda a derecha, cada piloto le corresponde la posición en la que ha quedado. Si Carlos Sainz ha quedado primero, le corresponderá el 0; esto es debido a que **los índices comienzan en 0**.
Podremos acceder a los elementos del interior de la matriz refiriéndonos a ellos por su índice.

```
13          /* Declaración 1 UNIDIMENSIONAL */
14          string[] matrizDeCadenas1 = new string[]
15          {
16              "Hola", "Mundo"
17          };
18
19          /* Declaración 2 UNIDIMENSIONAL */
20          string[] matrizDeCadenas2 = new string[3];
```

En la primera declaración definimos el tamaño de la matriz introduciendo los datos; de este modo sabemos que la matriz es de 2 elementos. En la segunda declaración indicamos que la matriz tiene 3 elementos y no hay nada en esos índices (nada que el usuario haya puesto).

1.1.2. Multidimensional

Si comprendemos la matriz unidimensional y el hecho de que una matriz es un tipo de dato, entonces podremos entender que una matriz multidimensional[3] es una matriz de matrices.

Índice	Valor	
0	Índice	Valor
	0	
	1	
	2	
1	Índice	Valor
	0	
	1	
	2	
2	Índice	Valor
	0	
	1	
	2	

Accedemos a cada elemento por cada uno de sus índices. Apréciese que no es posible que dos elementos tengan el mismo índice. También vemos que es una matriz de 3 dimensiones y por cada dimensión tenemos tres elementos En el siguiente ejemplo guardamos números en cada uno de las matrices:

Índice	Valor	
0	Índice	Valor
	0	50
	1	120
	2	100
1	Índice	Valor
	0	10
	1	80
	2	110
2	Índice	Valor
	0	60
	1	40
	2	130

Apréciese que si queremos acceder a un dato, mediante el índice de las matrices podemos acceder a él. Por ejemplo, si queremos saber lo que hay en el índice (1, 0) obtendremos «10».
Para plasmar esta matriz en código podremos hacerlo de las siguientes formas:

```
22          /* Declaración 1 MULTIDIMENSIONAL */
23          int[,] matrizMultidimensional1 = new int[,]
24              {
25                      { 50, 120, 100 },
26                      { 10, 80, 110 },
27                      { 130, 40, 60 }
28              };
29
30          /* Declaración 2 MULTIDIMENSIONAL */
31          int[,] matrizMultidimensional2 = new int[3,3];
32
33          matrizMultidimensional2[0, 0] = 50;
34          matrizMultidimensional2[0, 1] = 120;
35          matrizMultidimensional2[0, 2] = 100;
36
37          matrizMultidimensional2[1, 0] = 10;
38          matrizMultidimensional2[1, 1] = 80;
39          matrizMultidimensional2[1, 2] = 110;
40
41          matrizMultidimensional2[2, 0] = 60;
42          matrizMultidimensional2[2, 1] = 40;
43          matrizMultidimensional2[2, 2] = 130;
```

1.1.3. Escalonada (*jagged*)

Una matriz escalonada[4] es muy similar a la multidimensional, pero no es necesario que las matrices contenidas en la matriz sean todas de la misma dimensión.

Índice	Valor	
0	Índice	Valor
	0	50
	1	120
	2	100
	3	180
	4	60
	5	20
1	Índice	Valor
	0	10
	1	80
	2	110
2	Índice	Valor
	0	60
	1	40
	2	130
	3	130

En la figura superior tenemos una matriz que agrupa tres matrices, cada una de ellas de diferente longitud. Al igual que en la multidimensional,

accedemos a los datos mediante los dos índices; por ejemplo, en la posición (2,3) tenemos el número «130».
Dicha matriz se vería en código de las siguientes formas:

```
45          /* Declaración 1 ESCALONADA */
46          int[][] matrizEscalonada1 = new int[][]
47          {
48              new int[]
49              {
50                  50, 120, 100,
51                  180, 60, 20
52              },
53              new int[]
54              {
55                  10, 80, 110
56              },
57              new int[]
58              {
59                  60, 40, 130,
60                  130
61              }
62          };
63
64          /* Declaración 2 ESCALONADA */
65          int[][] matrizEscalonada2 = new int[3][];
66
67          matrizEscalonada2[0] =
68              new int[]
69              {
70                  50, 120, 100,
71                  180, 60, 20
72              };
73          matrizEscalonada2[1] =
74              new int[]
75              {
76                  10, 80, 110
77              };
78          matrizEscalonada2[2] =
79              new int[]
80              {
81                  60, 40, 130,
82                  130
83              };
```

1.2. Lista

Una lista[5] posee la misma función que una matriz, almacenar datos del mismo tipo. Con respecto de las matrices podemos destacar dos aspectos; uno favorable y otro desfavorable:

- No tenemos un tamaño fijo, pueden agrandarse cuanto se quiera.
- El tiempo de acceso a los valores aumenta debido a que no accedemos a los elementos mediante un índice. Esto es que las listas solo permiten un acceso secuencial a los elementos mientras que las matrices puede ser un acceso aleatorio.

Una lista consta, principalmente, de los siguientes datos:

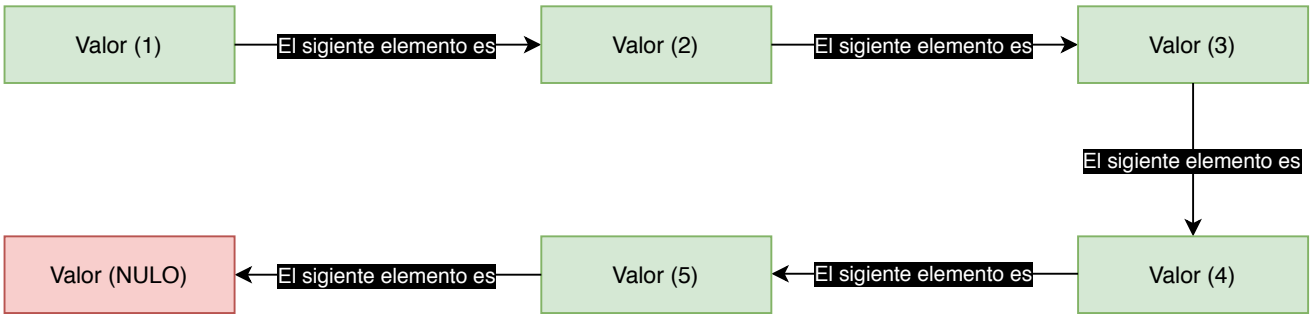
- Valor: este sera el valor que queremos guardar.
- Siguiente elemento: este campo será una referencia al elemento siguiente. Cada uno de los elementos de nuestra lista tendrá una referencia al elemento que le sigue. Para obtener cualquiera de los valores solo tenemos que conocer la posición del primer elemento e ir recorriendo de elemento en elemento.

Cabe destacar que una lista puede tener también un elemento que sea «Elemento anterior», y nos sirva para recorrer la lista en dos direcciones.
De hecho, tenemos varias posibilidades de listas que veremos en las siguientes subsecciones.
Pese a todos los tipos de listas, en *(C#* trabajaremos con una principalmente, $List<T>$. Para definir una lista de nombres podremos hacerlo tal que así:

```
13  List<string> lista = new List<string>();
14
15  lista.Add("Carlos Sainz");
16  lista.Add("Dani Sordo");
17  lista.Add("Colin McRae");
```

1.2.1. Simplemente enlazada

Cada elemento apunta al que le sigue o le precede. Lo normal es que apunte al que le sucede.



Como podemos apreciar, el último elemento apunta a nulo. Si queremos ir hasta el final de la lista habremos de empezar a recorrer la lista desde el primer elemento e ir «saltando» de elemento en elemento hasta que el elemento sea nulo; cuando sea nulo, sabremos que hemos llegado al final.
Desde un punto de vista de la memoria se podría ver de la siguiente manera (**atención, esto no implica que una lista sea guardada así en memoria**):

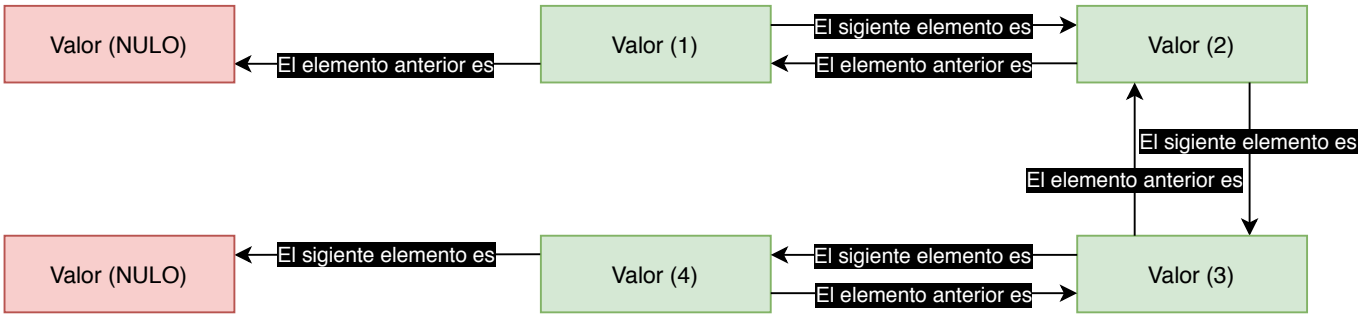
Posición de memoria	Elemento
0	Valor = 9 Dirección siguiente = 7
1	Valor = 7 Dirección siguiente = 2
2	Valor = 8 Dirección siguiente = ∅
3	Posición del primer elemento = 5
4	Valor = 5 Dirección siguiente = 6
5	Valor = 2 Dirección siguiente = 4
6	Valor = 1 Dirección siguiente = 0
7	Valor = 4 Dirección siguiente = 1

En la posición de memoria «3» tenemos guardada la posición del primer

elemento, y es lo único que conocemos de la lista. Nos está indicando que la lista empieza en la dirección «5», vayámonos a ella. ¿Qué nos encontramos? Que en esta posición tenemos guardado el número 2, que es el dato que queríamos guardar; ahora nos vamos al siguiente elemento, cuya dirección de memoria esta guardada en la misma dirección que acabamos de comprobar: en nuestro caso, la siguiente dirección de memoria es la 4. En la 4 tenemos el valor 5 y el siguiente elemento es el 6. De este modo vamos recorriendo la lista hasta llegar a un elemento que la dirección del siguiente elemento sea nulo. Véase el caso del elemento en la dirección de memoria 2: accedemos a él por medio del elemento que hay en la posición 1, y el siguiente elemento es nulo, es decir, en este elemento (2) acaba la lista.

1.2.2. Doblemente enlazada

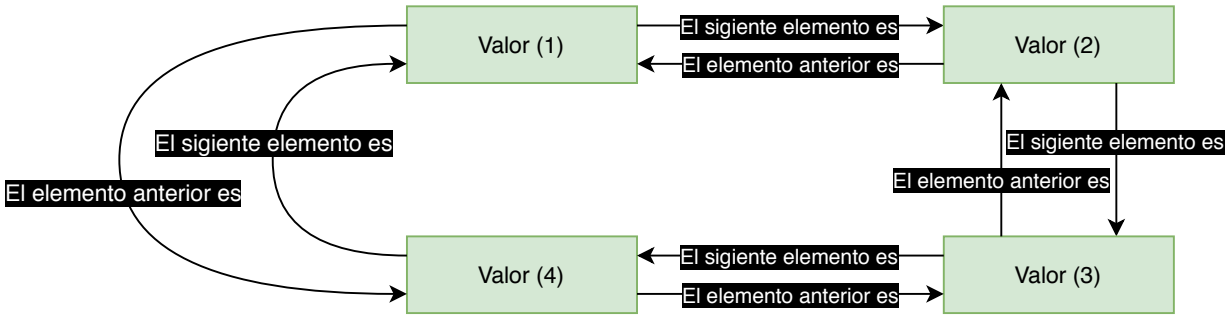
En este tipo de lista, cada elemento apunta a su siguiente y a su anterior.



De esta forma podremos recorrer la lista desde el primer elemento hasta el último y desde el último hasta el primero.

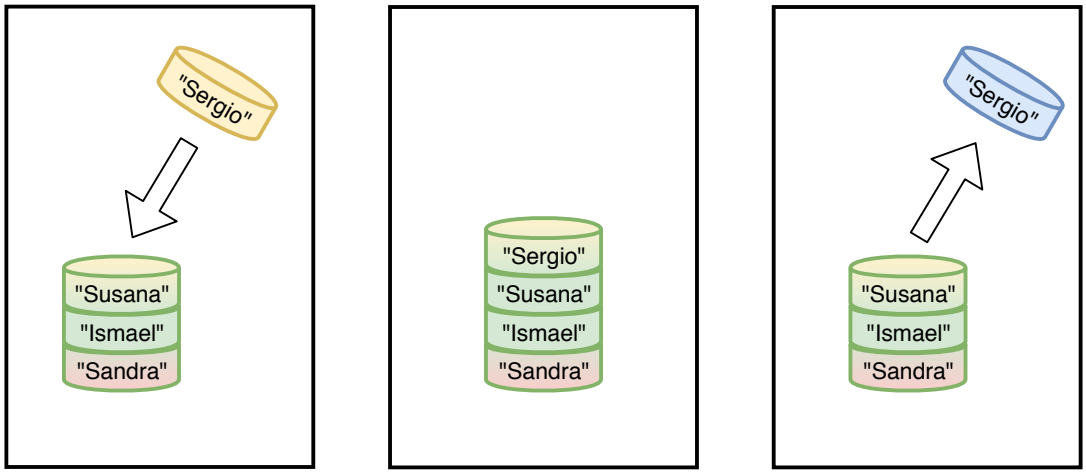
1.2.3. Circular

No vamos a entrar en detalles con este tipo de lista, pero su peculiaridad es que ninguno de los nodos apunta a nulo: el último apuntará al primero y, en caso de ser doblemente enlazada, el primero apuntará al último (con su puntero a «anterior»):



1.3. Pila (*stack*)

Las pilas[6] son similares a las listas, pero su peculiaridad es cómo se accede a sus datos. El modo de acceso es de tipo LIFO (*Last In First Out*), es decir, el último en entrar es el primero en salir.



En el ejemplo vemos una lista simple de cadenas. El primer elemento que se añadió es «"Sandra"», seguido de «"Ismael"» y el último añadido es «"Susana"»; Añadimos un elemento más, «"Sergio"», de tal forma que ahora «"Susana"» no está en la cima de la pila. En la última imagen sacaremos el último elemento añadido, que es «"Sergio"»; de este modo, el siguiente elemento a retirar de la pila es «"Susana"».
El siguiente código es una demostración de la imagen.
Creación de la pila:

```
10          Stack<string> pila = new Stack<string>
11              (
12                  new string[] { "Sandra", "Ismael", "Susana" }
13              );
```

Adición de la cadena «“Sergio”» a la pila:

```
15          pila.Push("Sergio");
```

Quitamos el último elemento («“Sergio”») y lo mostramos; con la última línea solo mostramos el último elemento.

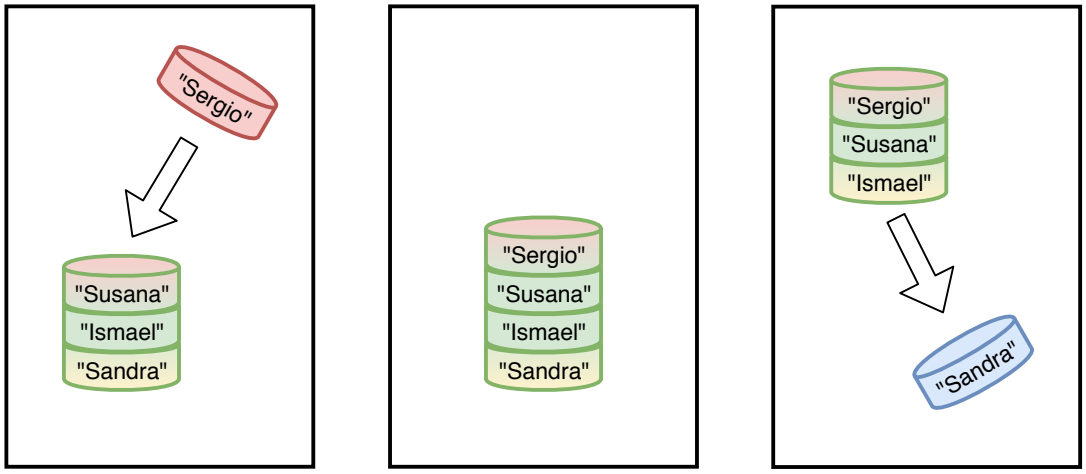
```
17          Console.WriteLine(pila.Pop());
18          Console.WriteLine(pila.Peek());
```

De este modo, tendríamos este programa:

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace Programas
5  {
6      class Program
7      {
8          static void Main()
9          {
10             Stack<string> pila = new Stack<string>
11                 (
12                     new string[] { "Sandra", "Ismael", "Susana" }
13                 );
14
15             pila.Push("Sergio");
16
17             Console.WriteLine(pila.Pop());
18             Console.WriteLine(pila.Peek());
19         }
20     }
21 }
```

1.4. Cola (*queue*)

Las colas[7] poseen la misma peculiaridad que las pilas: el acceso a los datos es especial. El modo de acceso a datos es FIFO (*First In First Out*), el primero en entrar será el primero en salir.



Vemos como el primer elemento que se ha añadido es «"Sandra"», por lo tanto dará igual cuántas cosas añadamos, el primero en salir será «"Sandra"». Pese a añadir «"Sergio"», «"Sandra"» saldrá la primera, como en la ultima figura indica. Una vez extraída de la cola, el siguiente elemento que le sigue para ser retirado es «"Ismael"».
Veamos un ejemplo
Creación de la cola:

```
10          Queue<string> cola = new Queue<string>
11              (
12                  new string[] { "Sandra", "Ismael", "Susana" }
13              );
```

Adición de la cadena «"Sergio"» a la pila. Este elemento será, de momento, el último en salir:

```
15          cola.Enqueue("Sergio");
```

Quitamos el último elemento («"Sandra"») y lo mostramos; con la última línea solo mostramos el último elemento («"Ismael"»).

```
17          Console.WriteLine(cola.Dequeue());
18          Console.WriteLine(cola.Peek());
```

De este modo, tendríamos este programa:

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace Programas
5  {
6      class Program
7      {
8          static void Main()
9          {
10             Queue<string> cola = new Queue<string>
11                 (
12                     new string[] { "Sandra", "Ismael", "Susana" }
13                 );
14
15             cola.Enqueue("Sergio");
16
17             Console.WriteLine(cola.Dequeue());
18             Console.WriteLine(cola.Peek());
19         }
20     }
21 }
```

1.5. Diccionario

Un diccionario[8] es un tipo muy potente de almacenado de datos pues, al igual que la lista, no tenemos un límite establecido y además el diccionario nos permite agregar una clave por cada dato que añadamos. Dicha clave será de un tipo concreto (determinado por el usuario). Veamos un ejemplo para una mejor comprensión:

Clave	Valor
"Carlos Sainz"	26
"Colin McRae"	25
"Dani Sordo"	17

La funcionalidad es muy similar a la de una matriz unidimensional, con la salvaguarda de que lo que sería el índice ahora es una clave. En el ejemplo vemos un diccionario donde los nombres de los pilotos (cadena) son la clave y el valor que les corresponde a cada uno son el número de victorias (número entero). Si buscamos en nuestro diccionario el valor que hay en la clave «Colin McRae» nos dará 25.
Con la siguiente linea declaramos el diccionario:

```
13 Dictionary<string, int> diccionario = new Dictionary<string, int>();
```

Podemos agregar elementos de la siguiente forma:

```
15 diccionario.Add("Carlos Sainz", 26);
16 diccionario.Add("Colin McRae", 25);
17 diccionario.Add("Dani Sordo", 0);
```

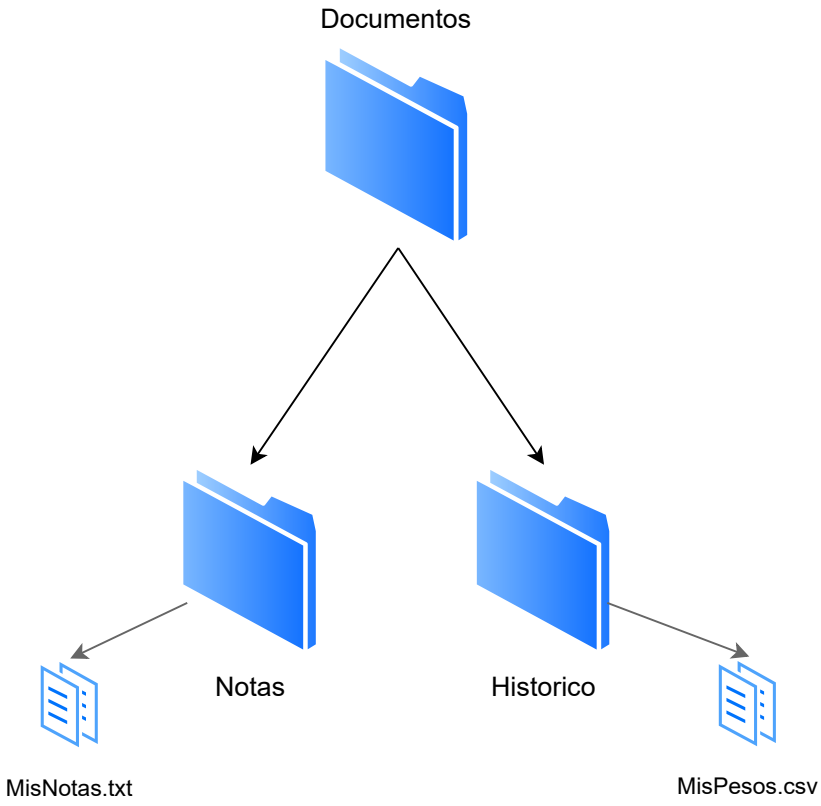
Véase que «"Dani Sordo"» no tiene las 17 victorias que le corresponden; para modificar un valor en una clave específica lo hacemos con indizadores:

```
19 diccionario["Dani Sordo"] = 17;
```

Capítulo 2

Manejo de directorios

Antes de ser capaces de entender cómo se manejan los ficheros, hemos de comprender que un fichero está contenido en un directorio[9].
Tanto en este capítulo como en el siguiente (Manejo de ficheros) trabajaremos con el siguiente esquema de directorios y ficheros:



Este será el sistema de archivos de una persona que quiera controlar su peso. Dentro de la carpeta «Documentos» tenemos dos carpetas: «Notas» e «Historico»; nótese que se refiere a «Hist**ó**rico» y por posibles problemas con las tildes hemos decidido suprimirla, pero como siempre digo: **la programación habría de realizarse toda en inglés**. Dentro del fichero «Notas» tendremos «MisNotas.txt», donde el usuario va a guardar notas en relación a alimentación (alimentos que engordan, hábitos saludables, horas para realizar las comidas, etc.). En el otro fichero, «Historico», guardaremos «MisPesos.csv», que será un fichero donde guardaremos de forma ordenada en un CSV con el día y el peso registrado.
No usaremos el explorador de archivos para la creación de ninguno de los elementos del árbol.
Comenzaremos por la creación de los directorios en el directorio de nuestro proyecto. Con las siguientes lineas creamos los directorios:

```
14          string ruta = Environment.CurrentDirectory + "\\Documentos";
15
16          Directory.CreateDirectory(ruta + "\\Notas");
17          Directory.CreateDirectory(ruta + "\\Historico");
```

En el caso de estar creados, «*CreateDirectory*» no sobreescribirá, simplemente no hará nada: están ya creados.
El resto de acciones que pueden hacerse con los directorios, tales como acceder a las propiedades, vienen en el MSDN.

Capítulo 3

Manejo de ficheros

C# nos proporciona un sistema de manejo de ficheros[10] sencillo, el cual desarrollaremos en las siguientes secciones.
Tenemos cinco tipos básicos en lo referente a manejo de ficheros:

- *Stream*[11]: lo usaremos para leer literalmente secuencias de bytes. **Es necesario para la lectura y la escritura de ficheros**.
- *StreamReader*[12]: podremos leer un fichero caracter a caracter con una codificación determinada.
- *StreamWriter*[13]: semejante al *StreamReader*; podremos escribir en un fichero siempre y cuando los datos a escribir estén codificados correctamente.
- *BinaryReader*[14]: como bien dice su nombre, lee datos escritos en binario en un fichero.
- *BinaryWriter*[15]: necesario para la lectura de ficheros binarios. Todos los ficheros son almacenados en binario, pero ello no implica que tengan una decodificación estándar. Podemos guardar 8 bits y cada uno de los valores de cada bit represente un dato concreto. Por ejemplo:
  AQUÍ IRÍA UNA FIGURA TO LO GUAPA POSIBLE DE UN FLAG DE UN VIDEOJUEGO QUE VA A QUEDAR FLAMAS

Enfocaremos todo el capítulo en el *StreamReader* y el *StreamWriter*, ya que funcionan del mismo modo que el *BinaryReader* y el *BinaryWriter*.

3.1. Apertura/creación del fichero

Habremos de cerciorarnos que el fichero que vamos a abrir para lo que sea (ya sea lectura, escritura o ambas) existe. En el caso de que no exista, tenemos que crearlo. Esto, sin embargo, *C#* lo hace muy cómodo ya que lo tiene muy automatizado todo; tenemos a nuestra disposición una serie de enumeraciones que nos servirán para un manejo de ficheros más potente:

1. *FileMode.Open*: podremos leer o escribir un fichero.
2. *FileMode.Append*: escribimos en un fichero; si no existe se crea.
3. *FileMode.Create*: se crea un nuevo fichero y se escribe; si el fichero existía se sobreescribe.
4. *FileMode.CreateNew*: se crea un nuevo fichero solo si no existe. En el caso de existir se lanzará una excepción.

3.2. Lectura/escritura de datos

Para el manejo de la mayoría de los tipos usaremos la cláusula «*using*»; con ella se simplifica la creación y nos olvidaremos de la destrucción.
Siguiendo el ejemplo de los directorios, vamos a escribir en «MisNotas.txt» el siguiente texto (que contendremos en una matriz de cadenas):

```
19  string[] misNotas = new string[]
20  {
21      "No cenar mucho",
22      "Evitar alcoholes y azúcares",
23      "Alejarse de bombones como yo",
24      "Comer fibra"
25  };
```

Lo escribimos en el fichero de la siguiente forma:

```
26  using (StreamWriter sw = File.AppendText(ruta+ @"\Notas\MisNotas.txt"))
27  {
28      foreach (string linea in misNotas)
29          sw.WriteLine(linea);
30  }
```

El otro fichero, «MisPesos.csv», se escribirá igual que el anterior. Evidentemente, al ser un CSV tenemos que respetar los separadores, que en nuestro caso serán punto y coma (más información en el capítulo Definiciones). Los datos a escribir los guardaremos en un diccionario:

```
32          Dictionary<DateTime, float> MisPesos = new Dictionary<DateTime, float>()
33          {
34              [new DateTime(2019, 06, 10)] = 65.20f,
35              [new DateTime(2019, 06, 11)] = 65.10f,
36              [new DateTime(2019, 06, 12)] = 65.30f,
37              [new DateTime(2019, 06, 13)] = 65.10f,
38              [new DateTime(2019, 06, 14)] = 64.90f,
39              [new DateTime(2019, 06, 15)] = 65.00f,
40              [new DateTime(2019, 06, 16)] = 64.90f,
41              [new DateTime(2019, 06, 17)] = 64.90f,
42              [new DateTime(2019, 06, 18)] = 64.80f
43          };
```

La clave será un objeto tipo *DateTime*, que será la fecha en la que el usuario se ha pesado, y el valor será un tipo *float*, el peso. Para escribir los datos, el procedimiento es el mismo que en la adición al archivo TXT, pero como es un CSV vamos a explicarlo un poco. En la primera línea de nuestro fichero irán los nombres de las columnas: «Fecha» y «Peso».

```
46              sw.WriteLine("Fecha;Peso");
```

Lo siguiente que introduciremos en el fichero serán los datos del diccionario:

```
47              foreach (KeyValuePair<DateTime, float> FechaYPeso in MisPesos)
48                  sw.WriteLine($"{FechaYPeso.Key.ToShortDateString()};{FechaYPeso.Value}");
```

De esta forma, tendríamos el siguiente fragmento:

```
44          using (StreamWriter sw = File.AppendText(ruta + @"\Historico\MisPesos.csv"))
45          {
46              sw.WriteLine("Fecha;Peso");
47              foreach (KeyValuePair<DateTime, float> FechaYPeso in MisPesos)
48                  sw.WriteLine($"{FechaYPeso.Key.ToShortDateString()};{FechaYPeso.Value}");
49          }
```

Lo bueno del CSV es que podemos abrir el fichero que se ha creado con una

hoja de cálculo y ver que los campos se respetan; la siguiente imagen está extraída directamente de una hoja de cálculo:

Fecha	Peso
10/06/2019	65,2
11/06/2019	65,1
12/06/2019	65,3
13/06/2019	65,1
14/06/2019	64,9
15/06/2019	65
16/06/2019	64,9
17/06/2019	64,9
18/06/2019	64,8

Los datos que hay en la tabla son exactamente los mismos que hemos introducido.
En el caso de querer leer uno de los ficheros, por ejemplo «MisNotas.txt», lo haríamos de la siguiente forma:

```
51          string lineaActual = "";
52          using (StreamReader sr = new StreamReader(ruta + @"MisNotas\MisNotas.txt"))
53          {
54              while ((lineaActual = sr.ReadLine()) != null)
55                  Console.WriteLine(lineaActual);
56          }
```

Lo curioso es la asignación, que se hace en el bucle.

```
54              while ((lineaActual = sr.ReadLine()) != null)
```

Primero asignamos a nuestra variable «lineaActual» lo que haya en la línea del fichero que corresponda. Una vez asignado, se evalúa si la cadena resultante no es nula; en el caso de serlo, nos saldríamos del bucle, y en el caso de que no, entraríamos para escribir por consola dicha línea.

```
55                  Console.WriteLine(lineaActual);
```

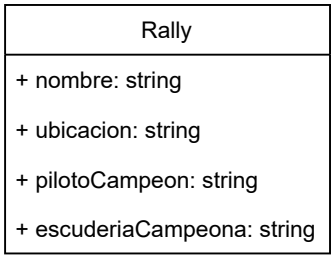
Capítulo 4

Programación orientada a objetos

Una correcta programación no es la que realiza la función requerida, sino la que lo hace lo más legible posible; para alcanzar este *Shambhala* de la programación tenemos que echar mano de nuestros amigos: los objetos. Es nuestra labor como programadores saber dividir un programa en partes lógicas, y cada una de esas partes será un objeto.
Muchas veces escucharemos la definición de «objeto» como una estructura con funciones y métodos, y en gran medida es eso, pero no es algo tan trivial. Un objeto consta, principalmente, de los siguientes elementos:

- Campos: serán los tipos de datos que contenga la clase.
- Funciones/Procedimientos: son métodos que van dentro de la clase.

Pero todo esto se ve mucho mejor con ejemplos; queremos hacer un objeto que sea «Rally». En el objeto irá el nombre del rally, la ubicación, el piloto campeón y la escudería campeona.



Haremos tres objetos «rally» y su contenido será el siguiente:

Rallyes

nombre	Campeonato de Europa de Rally
ubicacion	Europa
pilotoCampeon	Alexey Lukyanuk
escuderiaCampeona	ADAC Opel Rallye Junior Team

nombre	Campeonato Asia Pacífico de Rallyes
ubicacion	Asia
pilotoCampeon	Jan Kopecký
escuderiaCampeona	Škoda

nombre	Campeonato de África de Rally
ubicacion	África
pilotoCampeon	Manvir Baryan
escuderiaCampeona	Subaru

Ahora que ya sabemos que tenemos que crear los tres objetos «Rally», vamos a ver cómo se hace en código. Primeramente haremos una clase y en ella añadiremos los campos.

```
5       public string nombre { get; set; }
6       public string ubicacion { get; set; }
7       public string pilotoCampeon { get; set; }
8       public string escuderiaCampeona { get; set; }
```

Ya tendríamos los campos de la clase creados, ahora vayamos al constructor:

```
10      public Rally( string nombre, string ubicacion,
11                    string pilotoCampeon, string escuderiaCampeona)
12      {
13          this.nombre = nombre;
14          this.ubicacion = ubicacion;
15          this.pilotoCampeon = pilotoCampeon;
16          this.escuderiaCampeona = escuderiaCampeona;
17      }
```

Y la clase se vería tal que así:

```
namespace POO
{
    class Rally
    {
        public string nombre { get; set; }
        public string ubicacion { get; set; }
        public string pilotoCampeon { get; set; }
        public string escuderiaCampeona { get; set; }

        public Rally(  string nombre, string ubicacion,
                       string pilotoCampeon, string escuderiaCampeona)
        {
            this.nombre = nombre;
            this.ubicacion = ubicacion;
            this.pilotoCampeon = pilotoCampeon;
            this.escuderiaCampeona = escuderiaCampeona;
        }
    }
}
```

En el programa principal haremos una colección de rallyes; en nuestro caso, haremos una lista; podemos introducir los rallyes en la lista de diferentes maneras.
Añadiéndolo directamente en la creación de la lista:

```
            List<Rally> rallyes = new List<Rally>()
            {
                new Rally( "Campeonato de África de Rally",
                           "África",
                           "Manvir Baryan",
                           "Subaru")
            };
```

Creando el objeto asignándolo a una variable y metiéndola en la lista:

```
21          Rally rallyAsia = new Rally("Campeonato Asia Pacífico de Rallyes",
22                                      "Asia",
23                                      "Jan Kopecký",
24                                      "koda");
25          rallyes.Add(rallyAsia);
```

Añadiéndolo a la lista instanciando el objeto sin pasar por una variable:

```
27          rallyes.Add(new Rally( "Campeonato de Europa de Rally",
28                                 "Europa",
29                                 "Alexey Lukyanuk",
30                                 "ADAC Opel Rallye Junior Team"));
```

Y de este modo tendríamos en nuestra lista los tres rallyes metidos.

Capítulo 5

Indizadores

Un indizador[16] nos va a permitir tratar como una matriz una clase o una estructura. Para acceder a un elemento de una matriz lo hacemos con el operador «[ ]»; si quisiésemos obtener el elemento que hay en el índice 3, lo haremos con «array[3]». Vamos a hacer una modificación en la clase «Rally»: en vez de guardar el actual piloto campeón y su escudería, guardaremos un diccionario con una estructura de datos que se llamará «Equipo» como valor y un entero (*int*) como clave. La clave será el año en que el equipo ganó. La idea es acceder al equipo ganador con un indizador directamente sobre la clase. La estructura del equipo se vería tal que así:

```
7   public struct Equipo
8   {
9       public string Piloto;
10      public string Escuderia;
11
12      public Equipo(string piloto, string escuderia)
13      {
14          Piloto = piloto;
15          Escuderia = escuderia;
16      }
17  }
```

Ahora definamos el comportamiento del operador «[ ]» en nuestra clase. Lo haremos de la siguiente manera:

```
30      public Equipo this[int año]
31      {
32          get { return historico[año]; }
33      }
```

Con esto obtendríamos el equipo ganador del año que se introdujera entre corchetes.
Y la clase pasaría a ser esta:

```
using System.Collections.Generic;

namespace Indizadores
{
    class Rally
    {
        public struct Equipo
        {
            public string Piloto;
            public string Escuderia;

            public Equipo(string piloto, string escuderia)
            {
                Piloto = piloto;
                Escuderia = escuderia;
            }
        }

        public string nombre { get; set; }
        public string ubicacion { get; set; }
        public Dictionary<int, Equipo> historico { get; set; }

        public Rally(string nombre, string ubicacion, Dictionary<int, Equipo> historico)
        {
            this.nombre = nombre;
            this.ubicacion = ubicacion;
            this.historico = historico;
        }

        public Equipo this[int año]
        {
            get { return historico[año]; }
        }
    }
}
```

En el programa principal podremos acceder a la escudería y el piloto que ganó en un rally un año específico de una manera simple, sencilla y «bonita». Para añadir cada uno de los palmareses de las competiciones haremos tres diccionarios, uno para cada competición:

```
13          Dictionary<int, Rally.Equipo> historicoAfrica = new Dictionary<int, Rally.Equipo>
14          {
15              [2013] = new Rally.Equipo("Jassy Singh", "Subaru"),
16              [2014] = new Rally.Equipo("Gary Chaynes", "Mitsubishi"),
17              [2015] = new Rally.Equipo("Jaspreet Singh Chatthe", "Mitsubishi"),
18              [2016] = new Rally.Equipo("Don Smith", "Subaru"),
19              [2017] = new Rally.Equipo("Manvir Baryan", "koda"),
20          };
21
22          Dictionary<int, Rally.Equipo> historicoAsia = new Dictionary<int, Rally.Equipo>
23          {
24              [2013] = new Rally.Equipo("Gaurav Gill", "koda"),
25              [2014] = new Rally.Equipo("Jan Kopecký", "koda"),
26              [2015] = new Rally.Equipo("Pontus Tidemand", "koda"),
27              [2016] = new Rally.Equipo("Gaurav Gill", "koda"),
28              [2017] = new Rally.Equipo("Gaurav Gill", "koda"),
29              [2018] = new Rally.Equipo("Yuya Sumiyama", "koda"),
30          };
31
32          Dictionary<int, Rally.Equipo> historicoEuropa = new Dictionary<int, Rally.Equipo>
33          {
34              [2013] = new Rally.Equipo("Jan Kopecký", "koda"),
35              [2014] = new Rally.Equipo("Esapekka Lappi", "koda"),
36              [2015] = new Rally.Equipo("Kajetan Kajetanowicz", "Ford"),
37              [2016] = new Rally.Equipo("Kajetan Kajetanowicz", "Ford"),
38              [2017] = new Rally.Equipo("Kajetan Kajetanowicz", "Ford"),
39              [2018] = new Rally.Equipo("Alexey Lukyanuk", "Ford"),
40          };
```

Y los meteremos todos en nuestra lista de rallyes:

```
42          List<Rally> rallyes = new List<Rally>()
43          {
44              new Rally( "Campeonato de África de Rally",
45                         "África",
46                         historicoAfrica),
47              new Rally( "Campeonato Asia Pacífico de Rallyes",
48                         "Asia",
49                         historicoAsia),
50              new Rally( "Campeonato de Europa de Rally",
51                         "Europa",
52                         historicoEuropa)
53          };
```

Y ya podríamos acceder a los ganadores de los rallyes mediante un indizador. Lo usaríamos de la siguiente manera:

```
55          Console.WriteLine( $"Ganador del rally en el año 2015:\n" +
56                             $"\tPiloto: {rallyes[0][2015].Piloto}\n" +
57                             $"\tEscudería: {rallyes[0][2015].Escuderia}");
```

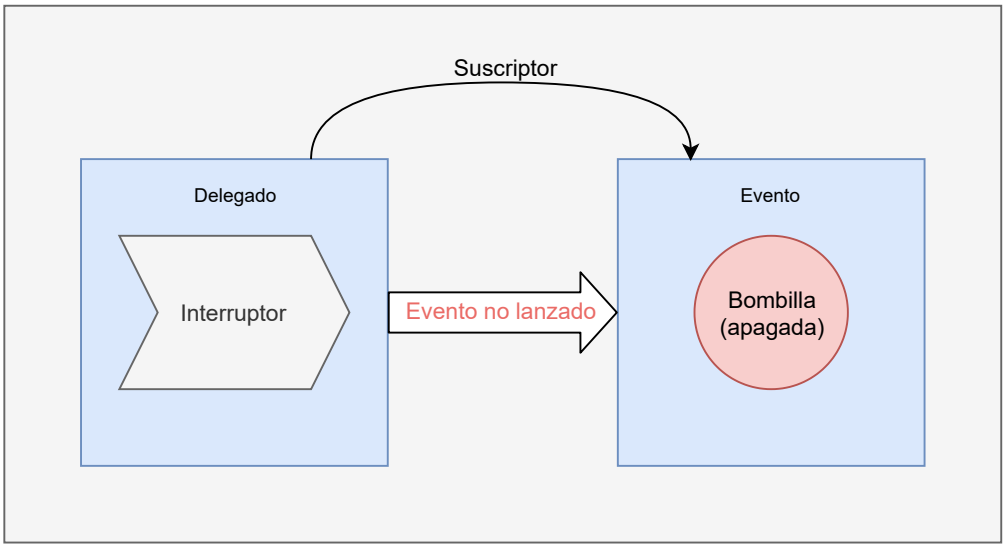
Obtenemos la siguiente salida:

```
Ganador del rally en el año 2015:
    Piloto: Jaspreet Singh Chatthe
    Escudería: Mitsubishi
```

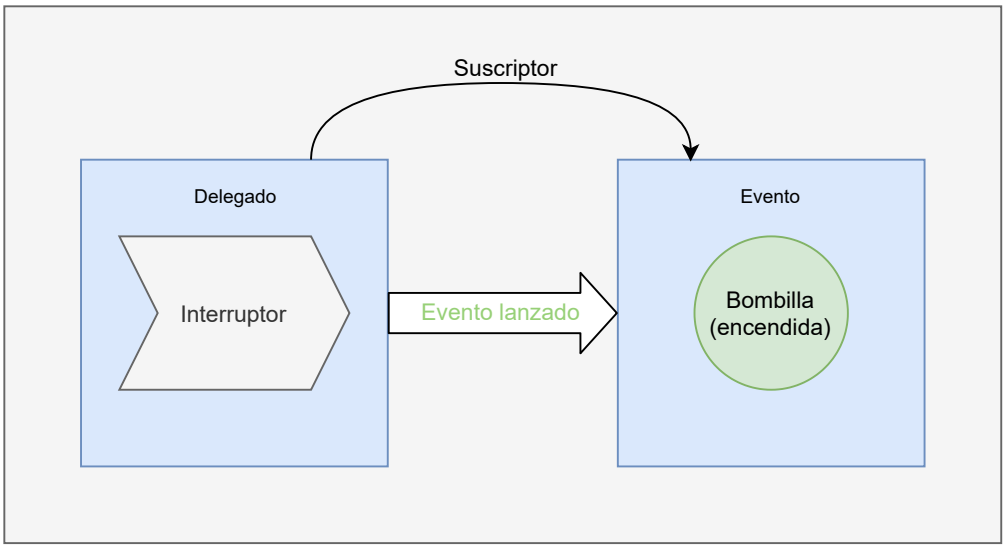
Capítulo 6

Eventos

Un evento[17] no es más que el envío de un mensaje cuando se produce una acción. Dicha acción puede ser causada por el usuario o por otro motivo ajeno a él. Quién provoca el evento es denominado «emisor del evento» y lo capturará un manejador de eventos, el cuál le habremos indicado que sea quién maneje el evento que acaba de ser lanzado.



El evento que tenemos en la figura superior está esperando que alguien lo llame, y el interruptor está esperando que alguien lo accione para que pueda lanzar el mensaje (lanzar el evento).



Una vez el interruptor se accione el evento se ejecutará porque habrá sido suscrito en el programa. Una suscripción no es más que «capturar» esa señal mandada por quién emite el mensaje y una vez capturada ejecutar una función.
Hagamos un programa simple: tendremos la bombilla, la cual se nos mostrará su estado en pantalla, y un interruptor, el cual el usuario podrá activar con la pulsación de la letra «R» y una vez activo el estado de la bombilla pasará a encendido. Como tenemos el estado de la bombilla en pantalla, veremos cuándo está activada.
Crearemos la clase bombilla y en ella definiremos el evento; en el programa principal suscribiremos ese evento a la función que se ejecutará cuando se lance. La clase tendrá un solo campo:

```
7       public delegate void Interruptor(Bombilla bombilla);
8       public event Interruptor interruptorPresionado;
```

El programa transcurrirá en el siguiente método, dentro de «Bombilla»:

```
12      public void ComenzarAEscuchar()
13      {
14          while (true)
15          {
16              Console.WriteLine(activo ? "Encendida" : "Apagada");
17              switch (Console.ReadKey(true).Key)
18              {
19                  case ConsoleKey.R:
20                      interruptorPresionado?.Invoke(this);
21                      break;
22                  default:
23                      break;
24              }
25              Console.Clear();
26          }
27      }
```

De este modo nos será más fácil lanzar el evento; lo lanzaremos en la línea 20.
El programa principal es muy simple:

1. Creamos una bombilla
2. Creamos una función que se adecúe al delegado que habiamos creado
3. Suscribimos el evento de la bombilla a la función que acabamos de crear
4. Llamamos al método que hemos creado en bombilla

El programa se verá tal que así:

```
1  namespace Eventos
2  {
3      class Program
4      {
5          static void Main()
6          {
7              Bombilla bombilla = new Bombilla();
8
9              bombilla.interruptorPresionado += EncenderBombilla;
10
11             bombilla.ComenzarAEscuchar();
12         }
13
14         static void EncenderBombilla(Bombilla bombilla)
15         {
16             bombilla.activo = true;
17         }
18     }
19 }
```

- Línea 7: creación del objeto «Bombilla»
- Línea 14-17: creación del método
- Línea 9: suscripción del método al evento
- Línea 11: llamada a la función de «Bombilla»

Capítulo 7

Tareas

Una tarea

Definiciones

- Puntero: un puntero puede parecer algo muy complejo de entender. De hecho, en *C#* no tenemos el tipo de datos «puntero» (por suerte o por desgracia). Un puntero no es más que una dirección de memoria.
  AQUÍ UNA FIGURA DE ESAS QUE TÚ Y YO SABEMOS QUE ESTÁN FLAMAS

- Nodo: dado que viene referido de una estructura de datos, un nodo es un registro dónde se tiene guardado un dato de interés y al menos un puntero que apunte a otro nodo colindante (que sea anterior o posterior).

- Directorio: agrupación virtual de ficheros y directorios. Un directorio puede no contener nada, pero un fichero siempre debe estar contenido en un directorio.
  FIGURA FLAMANTE DE UN ÁRBOL DE DIRECTORIOS FUA 3.0

- Fichero/archivo: conjunto de bits que representan información.

- Bit: unidad mínima de información. Un bit tiene dos estados: «0» y «1». Los bits van contenidos en bytes.

- Byte: Un byte es una agrupación de 8 bits independientes donde el valor de cada bit es esencial para la correcta comprensión de los datos. Del mismo modo que se agrupan los bits para guardar información, se pueden agrupar bytes para poder guardar más aún. Pongamos el ejemplo de un numero entero con signo (*int*) que se representa con cuatro bytes:

	Bit (0)	Bit (1)	Bit (2)	Bit (3)	Bit (4)	Bit (5)	Bit (6)	Bit (7)
Byte (0)	1	0	0	1	1	0	1	1
Byte (1)	0	0	1	1	0	0	0	1
Byte (2)	1	0	1	0	1	0	0	0
Byte (3)	1	0	0	1	0	1	1	1

El número representado en la figura superior sería el *10011011 00110001 10101000 10010111*$_2$ en binario y el *2603722903*$_{10}$ en decimal. Cada uno de los bits tiene una función trascendental en la información que queremos transmitir. Si alteramos un solo bit el número (o dato) que obtengamos será completamente diferente. Por ejemplo, cambiamos el último bit del tercer byte por un «1» (*10011011 00110001 10101001 10010110*$_2$); nos daría «*2603723158*$_{10}$», un número completamente diferente.

- Codificación: **referido a la codificación de caracteres**, la codificación es la conversión de un carácter (que nosotros consideramos «lenguaje natural», como por ejemplo la letra «G») a un símbolo de otro sistema de representación; como estamos en el campo de la informática, se codifica como un número. Véase ASCII.
- ASCII:
- Directorio del proyecto:
- CSV:

Bibliografía

[1] "Microsoft Windows MSDN - matriz." https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/arrays/.

[2] "Microsoft Windows MSDN - matriz unidimensional." https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/arrays/single-dimensional-arrays.

[3] "Microsoft Windows MSDN - matriz multidimensional." https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/arrays/multidimensional-arrays.

[4] "Microsoft Windows MSDN - matriz escalonada." https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/arrays/jagged-arrays.

[5] "Microsoft Windows MSDN - lista." https://docs.microsoft.com/es-es/dotnet/api/system.collections.generic.list-1?view=netframework-4.8.

[6] "Microsoft Windows MSDN - pila." https://docs.microsoft.com/es-es/dotnet/api/system.collections.stack?view=netframework-4.8.

[7] "Microsoft Windows MSDN - cola." https://docs.microsoft.com/es-es/dotnet/api/system.collections.queue?view=netframework-4.8.

[8] "Microsoft Windows MSDN - diccionario." https://docs.microsoft.com/es-es/dotnet/api/system.collections.generic.dictionary-2?view=netframework-4.8.

[9] "Microsoft Windows MSDN - directorio." https://docs.microsoft.com/es-es/dotnet/api/system.io.directory?view=netframework-4.8.

[10] "Microsoft Windows MSDN - fichero." https://docs.microsoft.com/es-es/dotnet/api/system.io.file?view=netframework-4.8.

[11] "Microsoft Windows MSDN - stream." https://docs.microsoft.com/es-es/dotnet/api/system.io.stream?view=netframework-4.8.

[12] "Microsoft Windows MSDN - stream reader." https://docs.microsoft.com/es-es/dotnet/api/system.io.streamwriter?view=netframework-4.8.

[13] "Microsoft Windows MSDN - stream writer." https://docs.microsoft.com/es-es/dotnet/api/system.io.streamreader?view=netframework-4.8.

[14] "Microsoft Windows MSDN - binary reader." https://docs.microsoft.com/es-es/dotnet/api/system.io.binaryreader?view=netframework-4.8.

[15] "Microsoft Windows MSDN - binary writer." https://docs.microsoft.com/es-es/dotnet/api/system.io.binarywriter?view=netframework-4.8.

[16] "Microsoft Windows MSDN - indizador." https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/indexers/.

[17] "Microsoft Windows MSDN - evento." https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/event.