# Tema 11.2

Descargar estos apuntes en pdf o html

## Índice

# Eventos en CSharp

## Definición

Un evento es un mensaje que se envía de un objeto a otro cuando ocurre algún suceso que deseamos notificar. Por lo tanto, podemos decir que **es una forma de comunicación entre objetos**.

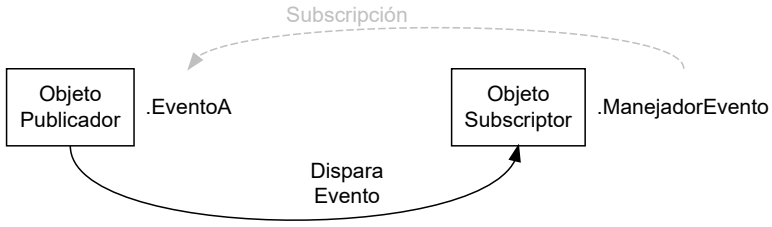
Dicho suceso puede ser debido a la interacción del usuario, como hacer clic en un botón, o podría proceder de cualquier otra lógica del programa, como el cambio del valor de una propiedad.

Más específicamente podremos diferenciar entre '*3 tipos*' de eventos:

1. Los que hemos mencionado y que me indicarán ua determinado suceso definido por el usuario, desencadenando una operación '*asíncrona*'.
   > 📌 **Nota:** Nosotros en este tema vamos a tratar estos últimos, asociados a un objeto delegado encargado de ejecutar esa operación asíncrona.
2. Los que me ayudarán a sincronizar varios hilos de ejecución. Esto es, un hilo concurrente comunica a otro o al principal que ya ha terminado de hacer una determinada tarea. Es una concreción del caso 1.
3. Generados por una ***'fuente'* de eventos** y que me servirán para recuperar el estado de un objeto o dominio en un determinado momento.
   > 📌 **Nota:** Estos dos últimos casos quedan completamente fuera del tema, pero son la base de muchas arquitecturas de Sotfware '*complejas*'.

## Nomenclatura básica

El **esquema más básico** y que vamos a tratar en este tema, es el siguiente:




- Los métodos asociados al delegado que se ejecutan al producirse un evento se denominan **manejadores o controladores**.
- El objeto que provoca el evento se conoce como **emisor o publicador** del evento y es el que contiene el tipo delegado.
- El objeto que maneja el evento se conoce como **receptor o suscriptor** del evento. Debe ser un objeto diferente al objeto emisor y es el que contiene los controladores.

# Gestión básica de eventos con CSharp

La sintaxis de definición será ...

```
class Publicador
{
    public event TipoDelegado IdEvento;
    ...
}
```

Cómo vemos, es similar a definir un campo o un propiedad de la clase. Pero... **¿Por qué utilizamos una palabra reservada event?**.

La idea es que se puedan distinguir de los elementos vistos hasta el momento en la definición de la clase. Por si no os habíais fijado, la ayuda del editor o IDE me distinguirá cada definición con un símbolo diferente.

Por ejemplo, para el siguiente código ...

```
class Publicador
{
    public TipoCampo IdCampo;
    public TipoPropiedad IdPropiedad { get; }
    public event TipoDelegado IdEvento;
}
```

Dependiendo del editor o IDE usado me mostrará junto a la definición un símbolo similar a los siguientes...

| Símbolo | Lo que mostrará el editor al ver las definiciones en un objeto de la clase `Publicador` |
|---|---|
| Caja azul | 🔹 `IdCampo` |
| Llave fija | 🔧 `IdPropiedad` |
| Rayo amarillo | ⚡ `IdEvento` |

De esta manea, el símbolo ⚡ nos indicará que el identificador `IdEvento` es un objeto delegado, que gestionará cierto evento del objeto y al que otros objetos se podrán suscribir con un manejador que cumpla la signatura del delegado.

Vamos a completar un esquema **Publicador - Subscriptor** con ejemplo genérico y posteriormente iremos concretando un poco más para entender el concepto.

Supongamos las siguientes definiciones esquemáticas genéricas...

```
// Clase que encapsulará información del evento.
// Solo la definiremos si hay información que pasar al subscriptor.
class DatosDelEvento
{
    public int Dato1 { get; }
    public string Dato2 { get; }
}

class Suscriptor
{
    public void ManejadorDeEventoDelSuscriptor(DatosDelEvento datos)
    {
        // Tratamiento del evento.
    }
}

class Publicador
{
    public delegate void ManejadorDeEvento(DatosDelEvento datos);
    public event ManejadorDeEvento IdEvento;

    public void OperacionQuePuedeDesencadenarElEvento()
    {
        // En algún momento el objeto publicador deberá notificar el evento.
        if (hayQueEnviarEvento)
        {
            // Generamos la multidifusión del evento a todos los objetos
            // subscriptores. Estos los tratarán a través del manejador que
            // asociaron.
            if (IdEvento != null)
                IdEvento(new DatosDelEvento(dato1, dato2));

            // Otra forma de expresar de forma más compácta el código de arriba
            // es usar el operador ?. de la siguiente forma.
            IdEvento?.Invoke(new DatosDelEvento(dato1, dato2));
        }
    }
}
```

En algún momento habrá en el programa un objeto publicador y uno o varios subscriptores instanciados....

```
Publicador p = new Publicador();

// En este esquema básico, por simplificar, los objetos suscriptores
// son del mismo tipo pero podrían ser de tipos diferentes.
Suscriptor s1 = new Suscriptor();
Suscriptor s2 = new Suscriptor();
```

Además, tendremos el siguiente código donde los objetos **suscriptores** `s1` y `s2` se suscriben al evento que genera el **publicador** `p`

```
p.IdEvento += s1.ManejadorDeEventoDelSuscriptor;
p.IdEvento += s2.ManejadorDeEventoDelSuscriptor;
```

Posteriormente, en algún lugar del programa, se ejecutará la siguiente línea con la operación que puede desencadenar el evento sobre el objeto publicador `p`.

```
...
// De producir la operación el evento, se ejecutará el código
// de los manejadores en los objetos subscriptores.
p.OperacionQuePuedeDesencadenarElEvento();
...
```

> ✋ **Importante:**
>
> El código de los manejadores se ejecutará por defecto en el mismo hilo de ejecución. Por esta razón, si el proceso de gestión del evento es muy costoso temporalmente podría bloquear la secuencia normal de ejecución.
>
> Aunque queda fuera de este tema, en estos casos convendría ejecutar el proceso del evento en un hilo aparte.

🎓 **Caso de estudio:**

Vamos a tratar de modelar mediante el mecanismo de comunicación con eventos entre objetos. Un modelo muy simplificado de una central nuclear donde:

Tendremos una clase `Reactor` a la que se le podrá cambiar la potencia que genera en `kW` a través de una propiedad. Esta potencia tendrá un máximo de `2000 kW` para que la temperatura del mismo se mantenga a menos de `1000ºC`. En este modelo simplificado, el valor de la propiedad `Temperatura` se calculará con una simple regla de 3 dados los máximos anteriormente dados.

El reactor generará un evento de cambio de temperatura al modificarse la potencia requerida en el mismo.

Piensa en cómo modelar dicha clase `Reactor` si no se te ocurre, puedes partir de la siguiente propuesta...

```
using System;
using System.Threading;

class Reactor
{
    public const int kW_MAXIMOS = 2000;
    public const double TEMPERATURA_MAXIMA = 1000D;

    public Reactor() => Potencia_kW = 0;

    // Temperatura es una propiedad calculada en función de la potencia.
    public double Temperatura => TEMPERATURA_MAXIMA * Potencia_kW / kW_MAXIMOS;

    // Definimos el evento de cambio de temperatura al que suscribirnos.
    public delegate void GestionTemperatura(double t);
    public event GestionTemperatura? CambioEnTemperatura;

    private int potencia_kW;
    public int Potencia_kW
    {
        get => potencia_kW;
        set
        {
            if (value != potencia_kW)
            {
                potencia_kW = value;
                // Si realmente se ha modificado el valor generamos el
                // evento de indicando la nueva temperatura.
                CambioEnTemperatura?.Invoke(Temperatura);
            }
        }
    }
}
```

Vamos ahora a modelas dos **clases suscriptoras** a los eventos de cambios de temperatura en el reactor. La primera será una sencilla clase `Alarma` de tal manera que, al instanciar un objeto `Alarma` le indicaremos un umbral de activación y además tendremos el **método manejador** `GestionActivacion` que cumplirá el interfaz del tipo delegado del evento de cambio de temperatura para poder suscribirse a él.

Piensa en cómo modelar dicha clase `Alarma` si no se te ocurre, puedes partir de la siguiente propuesta...

```
class Alarma
{
    public Alarma(double umbralActivacion) => UmbralActivacion = umbralActivacion;
    public bool Activada { get; private set; }
    private double UmbralActivacion { get; }
    public void GestionActivacion(double valor) => Activada = valor > UmbralActivacion;
}
```

El siguiente tipo que se suscribirá al evento del reactor será la clase `Refrigeración` . De tal manera que, una refrigeración se definirá con un numero de bombas de agua disponibles para hacer circular agua por la vasija del reactor y posteriormente llevarla al circuito de enfriamiento y evaporación. También recibirá una temperatura máxima para la que activar todas las bombas de agua, y de esta manera establecer el flujo de agua en el circuito.

Los objetos `Refrigeracion` se suscribirán al evento de cambio de temperatura en el reactor mediante el manejador `GestionTemperaturaReactor` que será el encargado de activar las bombas en el reactor. Para simplificar, vamos a implementar esta acción modificando una propiedad privada para modificación `BombasActivas` que se calculará también con una simple regla de tres, a partir del número de bombas disponibles y la temperatura a partir de la que se activarán todas.

Piensa en cómo modelar dicha clase `Refrigeracion` si no se te ocurre, puedes partir de la siguiente propuesta...

```
class Refrigeracion
{
    public Refrigeracion(int bombasDisponibles, double temperaturaMaxima)
    {
        TemperaturaMaxima = temperaturaMaxima;
        BombasDisponibles = bombasDisponibles;
        BombasActivas = 0;
    }

    public double TemperaturaMaxima { get; }
    public int BombasDisponibles { get; }
    public int BombasActivas { get; private set; }
    public void GestionTemperaturaReactor(double t)
    => BombasActivas = Convert.ToInt32(t * BombasDisponibles / TemperaturaMaxima);
}
```

Vamos generar un programa principal para probar que los eventos entre los objetos que modelan nuestro reactor se generan y se gestionan correctamente, para ello dentro de la clase **`Program`** voy a definir un método de utilidad, para visualizar lo que está sucediendo con los objetos involucrados. Por ejemplo, una posible implementación podría ser ...

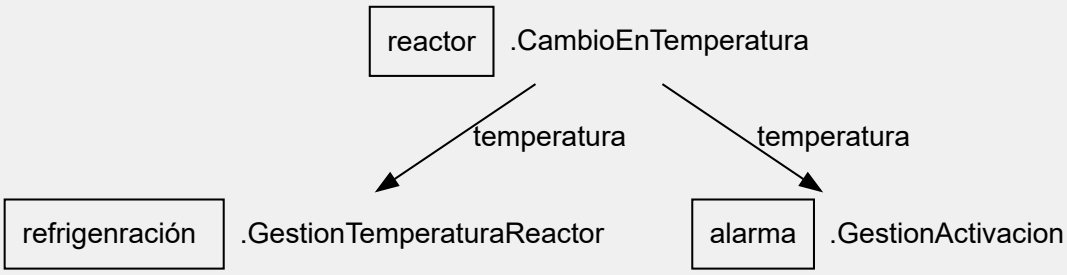
```
// Simula por consola un panel de motorización de los diferentes objetos que tenemos.
static void MuestraMonitorizacionControl(
                                Reactor reactor,
                                Refrigeracion refrigeracion,
                                Alarma alarma)
{
    string informacion =
    $"Potencia: {reactor.Potencia_kW, 9} kW" +
    $"\nTemperatura: {reactor.Temperatura, 6} ºC" +
    (alarma.Activada ? " ALARMA": $"{" ",7}") +
    $"\nBombas de refrigeración activas {refrigeracion.BombasActivas}";
    Console.SetCursorPosition(0, 0);
    Console.WriteLine(informacion);
}
```

Una posible implementación de un programa principal de prueba sería ...

```
static void Main()
{
    // Definimos los objetos relacionados.
    Reactor reactor = new Reactor();
    const int BOMBAS_DE_AGUA_REFRIGERACION = 3;
    Refrigeracion refrigeracion = new Refrigeracion(
                                  BOMBAS_DE_AGUA_REFRIGERACION,
                                  Reactor.TEMPERATURA_MAXIMA);
    Alarma alarma = new Alarma(Reactor.TEMPERATURA_MAXIMA);

    // Proceso de subscripción de la refrigeración y la alarma a los cambios
    // de temperatura en el objeto reactor.
    reactor.CambioEnTemperatura += refrigeracion.GestionTemperaturaReactor;
    reactor.CambioEnTemperatura += alarma.GestionActivacion;

    Random s = new Random();
    Console.Clear();
    for (int i = 0; i < 30; i++)
    {
        // Al cambiar la potencia solicitada al reactor de forma aleatoria
        // se desencadenará el evento de cambio de temperatura que puede
        // cambiar el estado de los objetos alarma y refrigeración.
        reactor.Potencia_kW = s.Next(300, Reactor.kW_MAXIMOS + 300);
        MuestraMonitorizacionControl(reactor, refrigeracion, alarma);
        Thread.Sleep(1000);
    }
}
```




✋ **Importante:** Si nos fijamos en el diseño consecuencia del uso de eventos, podemos observar que las clases '*desconocen*' con que otras clases se están relacionando. Por ejemplo, **`Alarma`** no sabe nada sobre quien le produce la activación. Solo sabe que se creó para saltar a partir de una determinado umbral y es lo que hace al recibir el evento. Podríamos rehusar este tipo **`Alarma`** para cualquier otro tipo que no sea una reactor.

Lo mismo sucede con la clase **`Reactor`** . No recibe ni usa ni se relaciona con el tipo **`Alarma`** o **`Refrigeracion`** y *`desconoce`* que tipo de objetos se suscribirán a sus eventos de cambio en la temperatura del reactor. Esto nos conduce a un fuerte **`desacoplamiento`** en nuestro diseño y a la posibilidad de hacer modificaciones o ampliaciones futuras sin tener que cambiar nuestra clase **`Reactor`** .

No todos los lenguajes dispondrán de la posibilidad de definir **eventos**, pero si que dispondrán de la posibilidad de implementar un esquema similar utilizando la programación orientada a objetos tradicional. A este tipo de esquema publicador/subscriptor simple, también se le conoce como **Patron Observador** y hemos visto que en C# se implementa de forma simple a través de **eventos**.

📌 **Nota1:** No debemos confundir el **patrón observador** con el **patrón publicador-subscriptor** que será una arquitectura mucho más compleja basada en el patrón observador.

📌 **Nota2:** El estudio en profundidad de ambos patrones y su comparación, quedan completamente fuera del alcance de este curso y lo único que se pretende en el tema es, entender el concepto de evento y gestionar en C# la comunicación simple entre objetos, utilizando los mismos.