

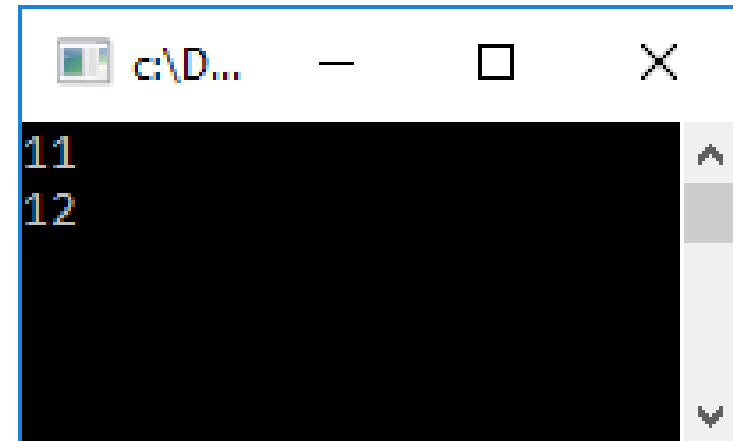
# C#.Net

# Delegados

- **Concepto:** Tipo especial de clase cuyos objetos **almacenan referencias a uno o mas métodos** de manera de poder ejecutar en cadena esos métodos.
- Permiten pasar **métodos como parámetros** a otros métodos
- Proporcionan un mecanismo para **implementar eventos**

# Delegados. Codifique

```
using System;
class Program{
    static void Main(){
        Console.WriteLine(sumaUno(10));
        Console.WriteLine(sumaDos(10));
        Console.ReadKey();
    }
    static int sumaUno(int n){
        return n+1;
    }
    static int sumaDos(int n){
        return n+2;
    }
}
```



# Delegados

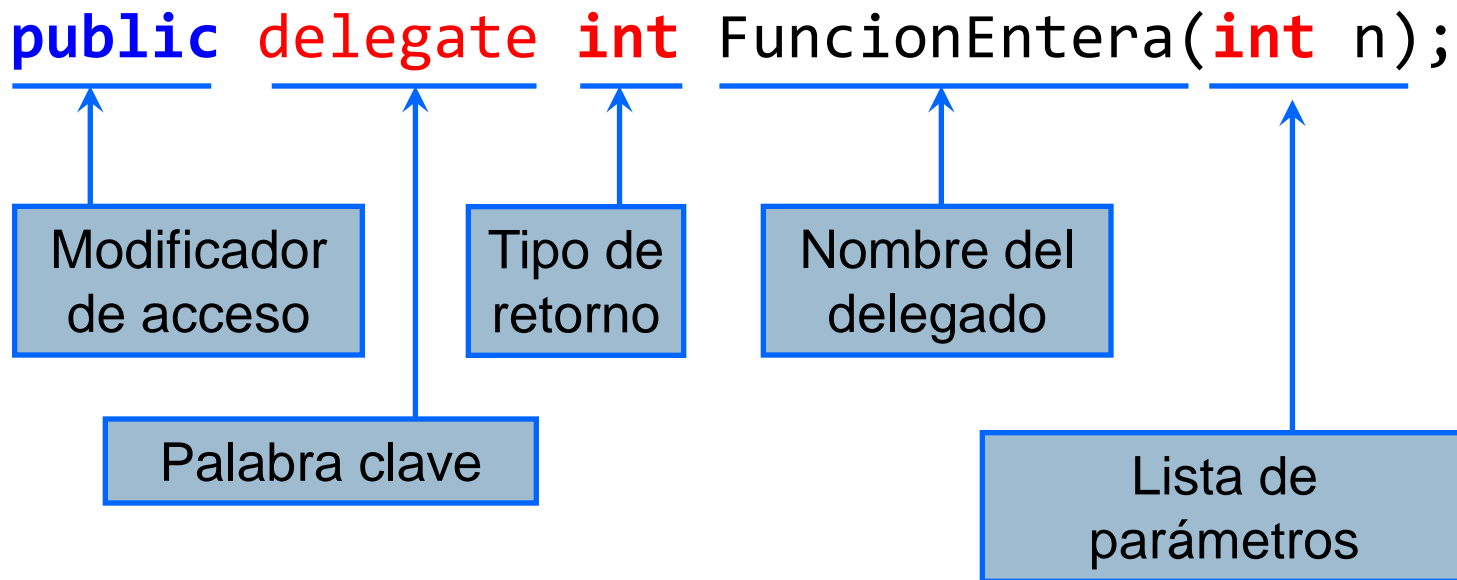
```
using System;
class Program{
    static void Main(){
        f = sumaUno;
        Console.WriteLine(f(10));
        f = sumaDos;
        Console.WriteLine(f(10));
        Console.ReadKey();
    }
    static int sumaUno(int n){
        return n+1;
    }
    static int sumaDos(int n){
        return n+2;
    }
}
```

Se desea poder invocar a los métodos **sumaUno** y **sumaDos** por medio de una variable **f**

**¿De qué tipo será **f**?**

# Delegados

## Sintaxis definición de un delegado



# Delegados. Codifique

```
using System;
delegate int FuncionEntera(int n);
class Program{
    static void Main(){
        Console.WriteLine(sumaUno(10));
        Console.WriteLine(sumaDos(10));
        Console.ReadKey();
    }
    static int sumaUno(int n){
        return n+1;
    }
    static int sumaDos(int n){
        return n+2;
    }
}
```

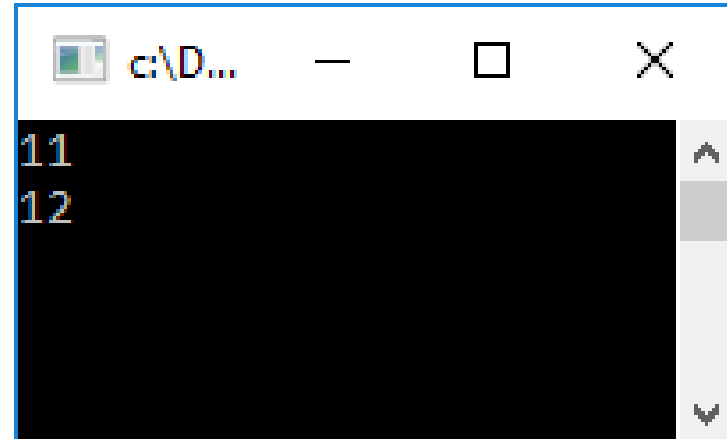
**FuncionEntera** es un tipo delegado que se corresponde con los métodos que reciben un parámetro de tipo **int** y que devuelven un valor de tipo **int**

# Delegados. Codifique

```
using System;
delegate int FuncionEntera(int n);
class Program{
    static void Main(){
        FuncionEntera f = sumaUno;
        Console.WriteLine(f(10));
        f = sumaDos;
        Console.WriteLine(f(10));
        Console.ReadKey();
    }
    static int sumaUno(int n){
        return n+1;
    }
    static int sumaDos(int n){
        return n+2;
    }
}
```

Se invoca sumaUno  
por medio de f

Se invoca sumaDos  
por medio de f

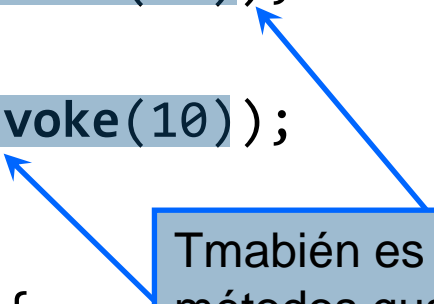


A screenshot of a Windows command prompt window. The title bar shows the path 'c:\D...' and standard window controls. The command prompt has a black background with white text. It displays the output of the program: '11' on the first line and '12' on the second line. A vertical scrollbar is visible on the right side of the window.

# Delegados. Codifique

```
using System;
delegate int FuncionEntera(int n);
class Program{
    static void Main(){
        FuncionEntera f = sumaUno;
        Console.WriteLine(f.Invoke(10));
        f = sumaDos;
        Console.WriteLine(f.Invoke(10));
        Console.ReadKey();
    }
    static int sumaUno(int n){
        return n+1;
    }
    static int sumaDos(int n){
        return n+2;
    }
}
```

También es posible invocar los métodos guardados en los delegados de forma explícita utilizando el método **Invoke**





# Delegados

- Las variables de tipo delegado pueden asignarse directamente con el nombre del método o con su correspondiente constructor pasando el método como parámetro.

```
f = sumaUno;
```

Es equivalente a:

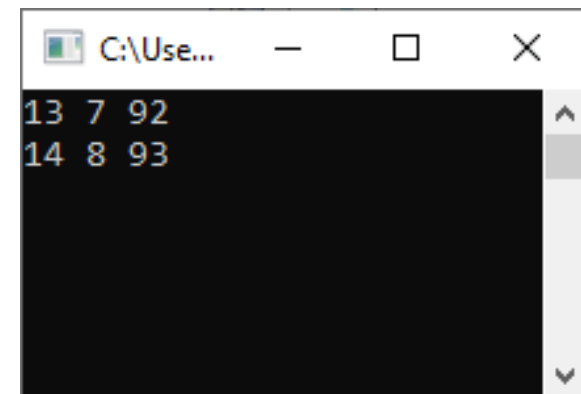
```
f = new FuncionEntera(sumaUno);
```

# Métodos pasados como parámetros. Codifique y ejecute

```
static void aplicar(int[] v, FuncionEntera f){  
    for(int i=0; i<v.Length; i++){  
        v[i] = f(v[i]);  
    }  
}
```

Agregue el método **aplicar** y reescriba el método **Main**

```
static void Main(){  
    int[] v = new int[] {11,5,90};  
    aplicar(v,sumaDos);  
    foreach(int i in v) Console.Write(i+" ");  
    aplicar(v,sumaUno);  
    Console.WriteLine();  
    foreach(int i in v) Console.Write(i+" ");  
    Console.ReadKey(true);  
}
```

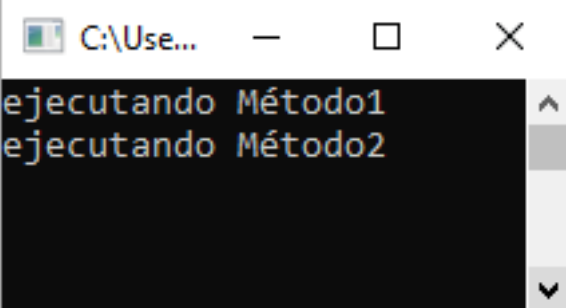


```
C:\Use...  
13 7 92  
14 8 93
```

# Codifique y ejecute

```
using System;
delegate void MetodoSinParametro();
class Program{
    static void Main(){
        MetodoSinParametro m;
        m=metodo1;
        m=m+metodo2; ←
        m();
        Console.ReadKey(true);
    }
    static void metodo1(){
        Console.WriteLine("ejecutando Método1");
    }
    static void metodo2(){
        Console.WriteLine("ejecutando Método2");
    }
}
```

Una variable de tipo delegado puede contener una lista de métodos que serán invocados secuencialmente

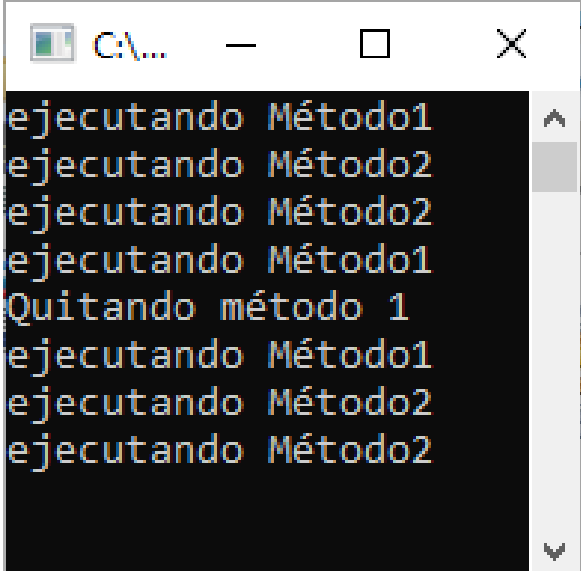


The screenshot shows a Windows console window with the title bar 'C:\Use...'. The console output displays two lines: 'ejecutando Método1' followed by 'ejecutando Método2', demonstrating the sequential execution of methods stored in the delegate variable.

# Codifique y ejecute

```
static void Main(){  
    MetodoSinParametro m;  
    m=metodo1;  
    m+=metodo2;  
    m+=metodo2;  
    m+=metodo1;  
    m();  
    Console.WriteLine("Quitando método 1");  
    m-=metodo1;  
    m();  
    Console.ReadKey(true);  
}
```

Modifique el método  
Main



The screenshot shows a console window with the following output:

```
ejecutando Método1  
ejecutando Método2  
ejecutando Método2  
ejecutando Método1  
Quitando método 1  
ejecutando Método1  
ejecutando Método2  
ejecutando Método2
```

# Notas

```
static void Main()
```

```
{
```

```
    MetodoSinParametro m = null;
```

```
    m = m + metodo1;
```

No hay error, el resultado de **null + metodo1** es **metodo1**

```
    m();
```

```
    m = m - metodo2;
```

No hay error al intentar quitar un método inexistente

```
    m();
```

```
    m = m - metodo1;
```

Quitar el último método de la lista de invocación devuelve **null**

```
    if (m==null)
```

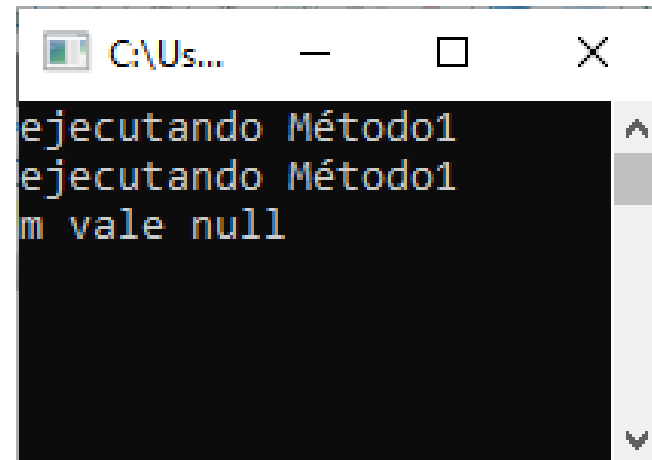
```
    {
```

```
        Console.WriteLine("m vale null");
```

```
    }
```

```
    Console.ReadKey(true);
```

```
}
```

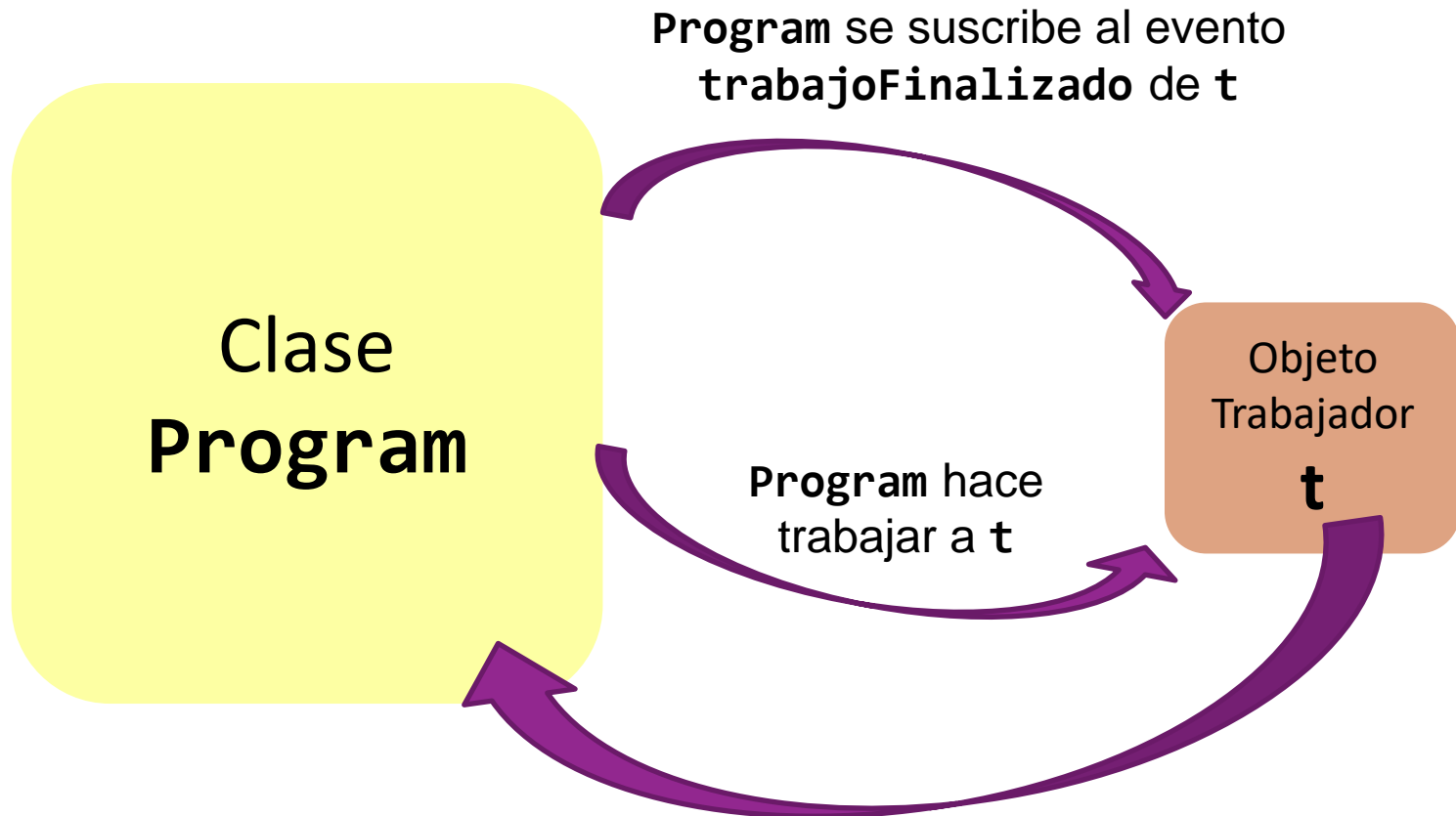


```
C:\Us...  
ejecutando Método1  
ejecutando Método1  
m vale null
```

# Eventos

- Cuando **ocurre** algo interesante, un objeto puede **notificarlo** a otras clases u objetos.
- La clase que **envía (o produce)** el evento recibe el nombre de **editor** y las clases que **reciben (o controlan)** el evento se denominan **suscriptores**.
- Propiedades de los evento:
  - El **editor** determina **cuándo** se produce un evento; los **suscriptores** determinan **qué operación** se realiza en respuesta al evento.
  - **Un evento** puede tener **varios suscriptores**. **Un suscriptor** puede controlar **varios eventos** de varios editores.
  - Nunca se provocan eventos que no tienen suscriptores.

# Utilizar delegados a modo de eventos – Ejemplo detallado

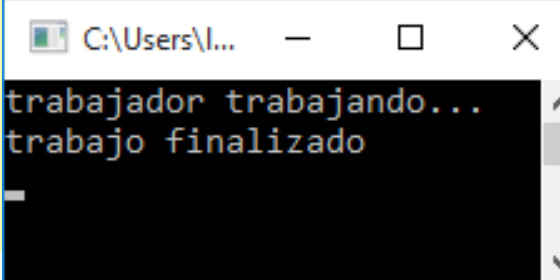


Cuando **t** termina su trabajo lanza el evento **TrabajoFinalizado** (avisando a **Program**, aunque realmente **t** no sabe a quién o quiénes está avisando)

# Utilizar delegados a modo de eventos – Ejemplo detallado

```
class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

La clase **Program** se suscribe al evento **TrabajoFinalizado** del objeto **t**, asignando su propio método **manejadorDelEvento** para manejar dicho evento



A screenshot of a Windows console window. The title bar shows the file path 'C:\Users\I...'. The console output displays two lines: 'trabajador trabajando...' followed by a new line, and then 'trabajo finalizado' on the next line. The cursor is positioned at the end of the second line.



# Utilizar delegados a modo de eventos – Ejemplo detallado

```
using System;
```

```
delegate void TrabajoFinalizadoEventHandler();
```

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}
```

Por convención suele agregarse el subfijo **EventHandler** al nombre del tipo delegado que se utiliza para implementar eventos

**TrabajoFinalizado** será un evento producido por un objeto **Trabajador**

Aquí se produce el evento invocando la lista de métodos encolados en el delegado. Si no se ha encolado ningún método la variable tiene el valor **null**, por eso es necesario verificarlo antes de intentar realizar la invocación.

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
         Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        ➡ t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

# Ejecución paso a paso


```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        ➡ t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```



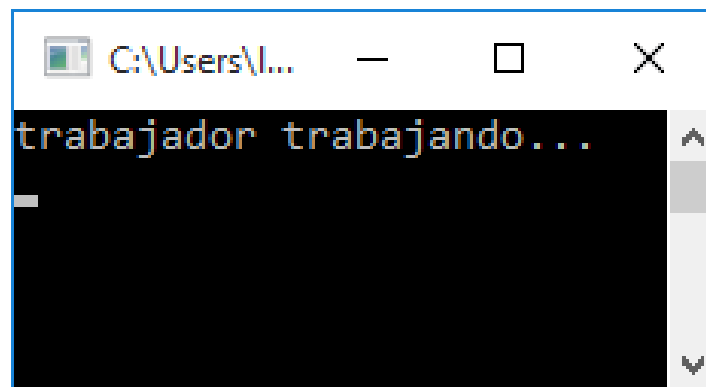
A diagram illustrating the execution flow. A bracket on the right side of the `Trabajar()` method in the `Trabajador` class is connected by a horizontal line to the `t.Trabajar();` call in the `Main()` method of the `Program` class. An arrow points from this line to the `Trabajar()` method, indicating the call site.

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        ➡ Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}
class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        ➔ Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}
class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```



# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```



# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
             TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }

    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

En la variable  
TrabajoFinalizado  
está encolado el método  
manejadorDelEvento

# Ejecución paso a paso

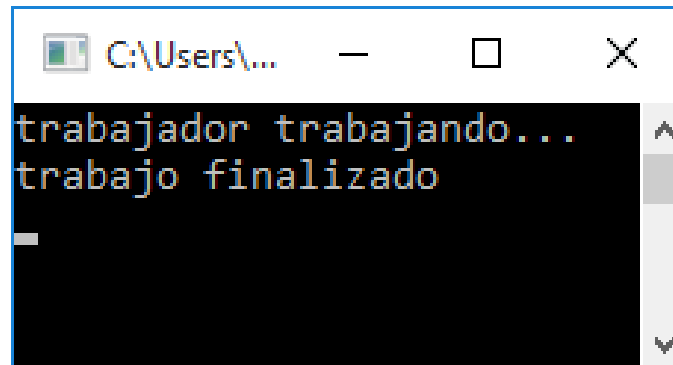
```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

A screenshot of a Windows console window. The title bar shows the file path "C:\Users\...". The console output displays two lines of text: "trabajador trabajando..." on the first line and "trabajo finalizado" on the second line. The text is in a monospaced font with some color coding (blue for "trabajador", green for "trabajo", red for "finalizado").

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}
class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```



# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```

# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}
class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```



# Ejecución paso a paso

```
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizado();
        }
    }
}

class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = manejadorDelEvento;
        t.Trabajar();
        ➡ Console.ReadKey();
    }
    private static void manejadorDelEvento()
    {
        Console.WriteLine("trabajo finalizado");
    }
}
```



# Ejemplo de una solución sin eventos

- **ESCENARIO 1:**

Se necesita codificar una clase **Ingresador** con un método público **Ingresar()** que permita al usuario ingresar líneas por la consola hasta que se ingrese un **string vacío** para que pueda ser utilizada de la siguiente manera:

```
class Program
{
    public static void Main(string[] args)
    {
        Ingresador ing=new Ingresador();
        ing.Ingresar();
    }
}
```

# Ejemplo de una solución sin eventos

```
class Ingresador
{
    public void Ingresar()
    {
        string st=Console.ReadLine();
        while(st != "")
        {
            st=Console.ReadLine();
        }
    }
}
```

# Ejemplo de una solución sin eventos

- **ESCENARIO 2:**

Ahora se necesita que al ingresarse una línea con **menos de 10 caracteres** se imprima en la consola la leyenda: “**Línea corta!**”.

Modifique el método Ingresar para satisfacer este requerimiento

# Ejemplo de una solución sin eventos

```
class Ingresador
{
    public void Ingresar()
    {
        string st=Console.ReadLine();
        while(st != "")
        {
            if (st.Length < 10)
            {
                Console.WriteLine("Línea corta!");
            }
            st=Console.ReadLine();
        }
    }
}
```

# Ejemplo de una solución sin eventos

- **ESCENARIO 3:**

Ahora se necesita que cada vez que ingrese un **número**, se imprima en la consola la leyenda: **“Número Ingresado!”** En caso contrario no se debe hacer nada.

Modifique el método Ingresar para satisfacer este requerimiento

# Ejemplo de una solución sin eventos

```
class Ingresador
{
    public void Ingresar()
    {
        string st=Console.ReadLine();
        while(st != "")
        {
            try
            {
                double d=double.Parse(st);
                Console.WriteLine("Número Ingresado!");
            } catch {}
            st=Console.ReadLine();
        }
    }
}
```

# Ejemplo con eventos

- Claramente la clase **Ingresador** de la solución anterior **no resulta lo suficientemente genérica** como para poder resolver sin ningún retoque los tres escenarios anteriores descritos.
- Una mejor solución sería que la clase **Ingresador** no asumiera la responsabilidad de tomar las acciones requeridas en los tres escenarios presentados, sino que **simplemente anunciara** la situación **produciendo los eventos adecuados**.
- De acuerdo a los requerimientos establecidos en los escenarios presentados la clase Program se suscribirá o no lo hará a estos eventos.

# Ejemplo con eventos

```
delegate void AnuncioEventHandler();
```

```
class Ingresador  
{
```

```
    public AnuncioEventHandler NumeroIngresado;  
    public AnuncioEventHandler LineaCortaIngresada;
```

```
    public void Ingresar()  
    {
```

```
        string st=Console.ReadLine();
```

```
        while(st != "") {
```

```
            if (st.Length < 10 && LineaCortaIngresada != null)
```

```
                LineaCortaIngresada();
```

```
            try
```

```
            {
```

```
                double d=double.Parse(st);
```

```
                if (NumeroIngresado != null) NumeroIngresado();
```

```
            } catch {}
```

```
            st=Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

Los delegados públicos pueden utilizarse a modo de eventos

Eventualmente se producen los eventos invocando los métodos encolados en los delegados

Es importante asegurarse que los delegados sean distinto de null antes de intentar invocar los métodos encolados.



# Ejemplo con eventos

## Escenario 1

```
class Program
{
    public static void Main(string[] args)
    {
        Ingresador ing=new Ingresador();
        ing.Ingresar();
    }
}
```

Simplemente la clase Program no necesita suscribirse a ningún evento producido por el objeto ing

# Ejemplo con eventos

## Escenario 2

```
class Program
{
    public static void Main(string[] args)
    {
        Ingresador ing=new Ingresador();
        ing.LineaCortaIngresada=ing_LineaCortaIngresada;
        ing.Ingresar();
    }
    private static void ing_LineaCortaIngresada()
    {
        Console.WriteLine("Línea Corta!");
    }
}
```

La clase Program se suscribirse al evento LineaCortaIngresada producido por el objeto ing

# Ejemplo con eventos

## Escenario 3

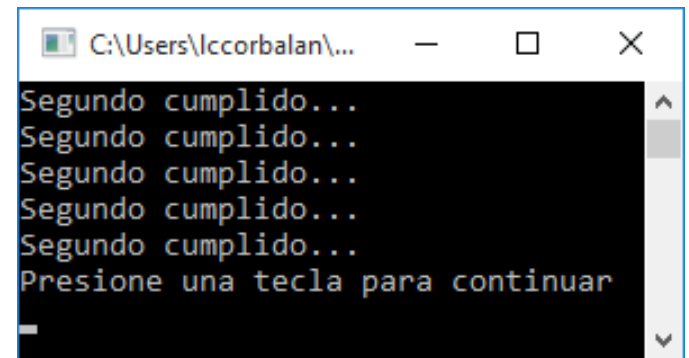
```
class Program
{
    public static void Main(string[] args)
    {
        Ingresador ing=new Ingresador();
        ing.NumeroIngresado=ing_NumeroIngresado;
        ing.Ingresar();
    }
    private static void ing_NumeroIngresado()
    {
        Console.WriteLine("Número Ingresado!");
    }
}
```

La clase Program se suscribirse al evento NumeroIngresado producido por el objeto ing

# Ejercicios con solución

# Ejercicio 1

- Complete el código implementado la clase `ContadorSegundos` que produce un evento `SegundoCumplido` cada vez que transcurre un segundo.



```
C:\Users\lccorbalan\...
Segundo cumplido...
Segundo cumplido...
Segundo cumplido...
Segundo cumplido...
Segundo cumplido...
Presione una tecla para continuar
```

Para frenar la ejecución durante un segundo utilice la sentencia:  
`System.Threading.Thread.Sleep(1000);`

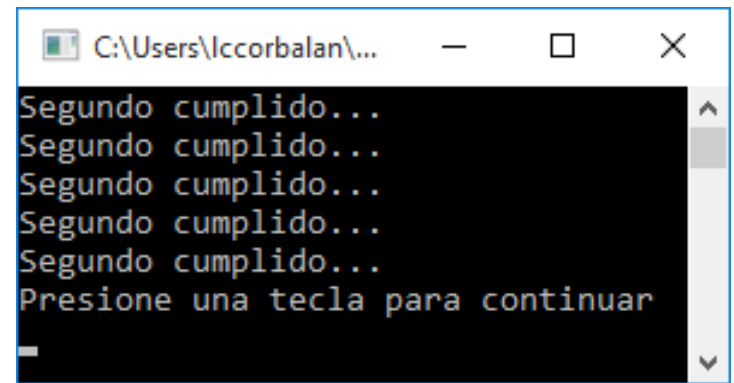
```
class Program
{
    public static void Main(string[] args)
    {
        ContadorSegundos cont = new ContadorSegundos();
        cont.Cantidad=5;
        cont.SegundoCumplido=cont_SegundoCumplido;
        cont.Contar();
        Console.WriteLine("Presione una tecla para continuar");
        Console.ReadKey();
    }
    private static void cont_SegundoCumplido()
    {
        Console.WriteLine("Segundo cumplido...");
    }
}
```

# Ejercicio 1 - Solución

```
class ContadorSegundos
{
    public int Cantidad {get;set;}
    public SegundoCumplidoEventHandler SegundoCumplido;

    public void Contar()
    {
        for(int i=1;i<=Cantidad;i++)
        {
            System.Threading.Thread.Sleep(1000);
            if (SegundoCumplido != null)
            {
                SegundoCumplido();
            }
        }
    }
}
```

```
delegate void SegundoCumplidoEventHandler();
```

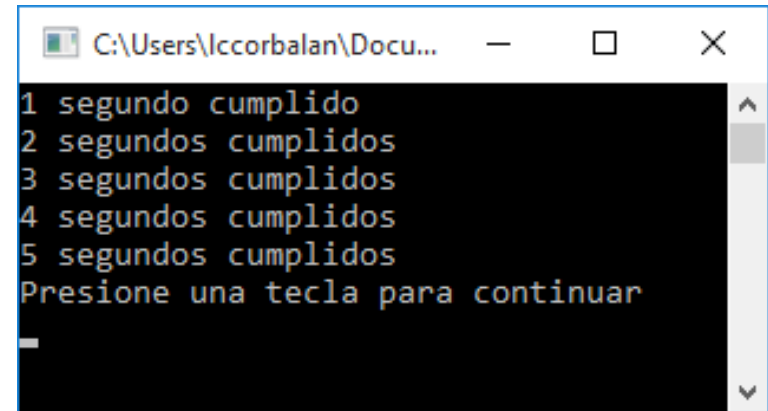


```
C:\Users\lccorbalan\...
Segundo cumplido...
Segundo cumplido...
Segundo cumplido...
Segundo cumplido...
Segundo cumplido...
Presione una tecla para continuar
```

# Ejercicio 2

Modifique el código de la clase ContadorSegundos para utilizarlo de esta manera

```
class Program
{
    public static void Main(string[] args)
    {
        ContadorSegundos cont = new ContadorSegundos();
        cont.Cantidad=5;
        cont.SegundoCumplido=cont_SegundoCumplido;
        cont.Contar();
        Console.WriteLine("Presione una tecla para continuar");
        Console.ReadKey();
    }
    private static void cont_SegundoCumplido(int n)
    {
        string leyenda = " segundo cumplido";
        if (n>1) leyenda = " segundos cumplidos";
        Console.WriteLine(n+leyenda);
    }
}
```



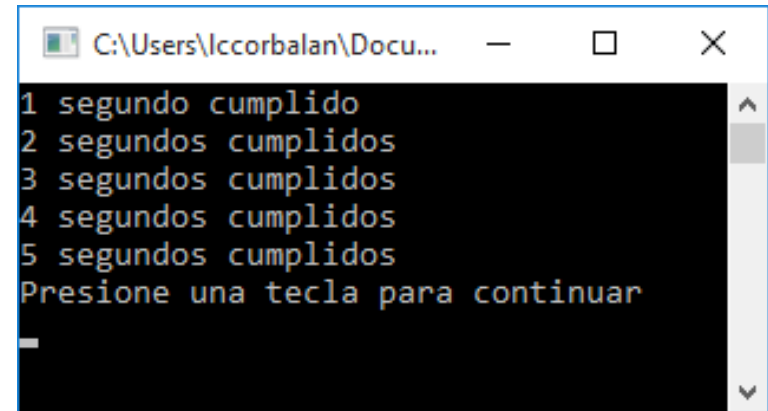
```
C:\Users\lccorbalan\Docu...
1 segundo cumplido
2 segundos cumplidos
3 segundos cumplidos
4 segundos cumplidos
5 segundos cumplidos
Presione una tecla para continuar
-
```

# Ejercicio 2 - Solución

```
class ContadorSegundos
{
    public int Cantidad {get;set;}
    public SegundoCumplidoEventHandler SegundoCumplido;

    public void Contar()
    {
        for(int i=1; i<=Cantidad; i++)
        {
            System.Threading.Thread.Sleep(1000);
            if (SegundoCumplido != null)
            {
                SegundoCumplido(i);
            }
        }
    }
}

delegate void SegundoCumplidoEventHandler(int n);
```



```
C:\Users\lccorbalan\Docu...
1 segundo cumplido
2 segundos cumplidos
3 segundos cumplidos
4 segundos cumplidos
5 segundos cumplidos
Presione una tecla para continuar
```



# Convenciones de nomenclatura

- Por convención, si es posible, los nombres que se usen para los eventos, los delegados y las clases de datos de evento deberían compartir una raíz común. Por ejemplo, si define un evento **CapacidadExcedida** en una de sus clases, el delegado se debería denominar **CapacidadExcedidaEventHandler** y los datos de evento se deberían denominar **CapacidadExcedidaEventArgs**.
- Para los nombres de los eventos se utilizarán preferentemente verbos en **gerundio** (ejemplo *IniciandoTrabajo*) o **participio** (ejemplo *TrabajoFinalizado*) según se produzcan antes o después del hecho de significación.

# Ejemplo respetando convención de nomenclatura

- Se implementará una clase **Ingresador** para introducir texto desde la consola repetidamente hasta que el usuario tipee el número 0 (cero)
- El ingresador provocará un evento **NumTipeado** cuando el texto introducido por el usuario se corresponda con un valor numérico válido. En caso contrario simplemente ignora la entrada y prosigue en el loop.
- La clase Program creará un objeto **Ingresador** y se suscribirá al evento **NumTipeado** de este objeto manejándolo con un método que recibirá un parámetro **NumTipeadoEventArgs** en cuya propiedad **Valor** se encontrará el número ingresado por el usuario

# Ejemplo - Resolución

```
delegate void NumTipeadoEventHandler (NumTipeadoEvenArgs e);
```

```
class Program{
```

```
    static void Main() {
```

```
        Ingresador ing = new Ingresador(); ←
```

Se crea un objeto Ingresador

```
        ing.NumTipeado = ingNumTipeado; ←
```

Se suscribe al evento

```
        ing.Ingresar(); ←
```

Se invoca el método Ingresar

```
    }
```

```
    static void ingNumTipeado (NumTipeadoEvenArgs e) { ←
```

```
        Console.WriteLine("Se ha ingresado {0}", e.Valor);
```

```
    }
```

```
}
```

Método con el cual Program se suscribió al evento. Cada vez que se produzca el evento NumTipeado se ejecutará este método. Es decir que es el método con el cual Program maneja el evento

# Ejemplo - Resolución

```
class NumTipeadoEventArgs:EventArgs {  
    public double Valor {get;set;}  
}
```

No es obligatorio, pero se recomienda que derive de EventArgs

```
class Ingresador {  
    public NumTipeadoEventHandler NumTipeado;  
    public void Ingresar() {  
        NumTipeadoEventArgs e = new NumTipeadoEventArgs() {Valor = -1};  
        while (e.Valor != 0) {  
            try{  
                e.Valor = double.Parse(Console.ReadLine());  
                if (NumTipeado != null)  
                    NumTipeado(e);  
            } catch {}  
        }  
    }  
}
```

Publica la variable NumTipeado para que puedan suscribirse al evento

Invoca todos los métodos encolados. Si no se ha encolado ningún método la variable tiene el valor null, por eso es necesario verificarlo antes de intentar realizar la invocación.

# Ejemplo 2

- Se desea modificar la clase **Ingresador** para poder controlar externamente la finalización del loop de ingreso de datos.
- El que se suscriba al evento tendrá también la posibilidad de finalizar la entrada de datos estableciendo convenientemente una propiedad del objeto **NumTipeadoEventArgs** recibido como parámetro
- Por lo tanto **NumTipeadoEventArgs** tendrá dos propiedades, una para alojar el valor numérico ingresado (**Valor**) y la otra para ser establecida por el suscriptor indicando el fin a la entrada de datos (**FinDeIngreso**)

# Ejemplo 2 - Resolución

```
class Program{  
    static void Main() {  
        Ingresador ing = new Ingresador();  
        ing.NumTipeado = ingNumTipeado;  
        ing.Ingresar();  
    }  
  
    static void ingNumTipeado (NumTipeadoEvenArgs e) {  
        Console.WriteLine("Se ha ingresado {0}",e.Valor);  
        if (e.Valor == 0)  
            e.FinDeIngreso = true;  
    }  
}
```

# Ejemplo 2 - Resolución

```
class NumTipeadoEvenArgs:EventArgs {  
    public double Valor {get;set;}  
    public bool FinDeIngreso {get;set;}  
}  
  
class Ingresador {  
    public NumTipeadoEventHandler NumTipeado;  
    public void Ingresar() {  
        NumTipeadoEvenArgs e;  
        e=new NumTipeadoEvenArgs() {FinDeIngreso=false};  
        while ( !e.FinDeIngreso ) {  
            try{  
                e.Valor = double.Parse(Console.ReadLine());  
                if (NumTipeado != null)  
                    NumTipeado(e);  
            }catch{}  
        }  
    }  
}
```

# Parámetro sender en los eventos

- Otra convención para los delegados asociados a eventos es la utilización de dos parámetros:
  - El primero, un objeto genérico (de la clase **object**) llamado **sender** utilizado para que el propio objeto que produce el evento se envíe a sí mismo
  - El segundo corresponde al argumento **EventArgs** o alguno derivado como ya hemos visto.
- Ejemplo:

```
delegate void AcontecimientoEventHandler(  
    object sender, EventArgs e);
```

  - Aunque el argumento **EventArgs** no posee propiedades ni comportamiento alguno, podría invocarse con cualquier objeto de una clase derivada.

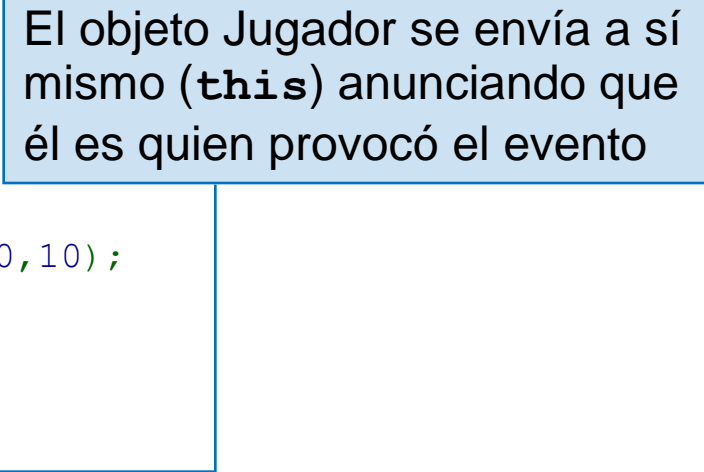


# Ejemplo

```
delegate void MovidoEventHandler(object sender, EventArgs e);
```

```
class Jugador
{
    static Random genAleatorio=new Random();
    int posicion=0;
    public char Id{get;set;}
    public MovidoEventHandler Movido;
    public void Mover()
    {
        posicion += Jugador.genAleatorio.Next(0,10);
        if (Movido != null)
        {
            Movido(this,new EventArgs());
        }
    }
    public void Imprimirse()
    {
        Console.WriteLine("Jugador {0} => posición {1}",Id,posicion);
    }
}
```

El objeto Jugador se envía a sí mismo (**this**) anunciando que él es quien provocó el evento




# Ejemplo

```
delegate void MovidoEventHandler(object sender, EventArgs e);

class Jugador
{
    static Random genAleatorio=new Random();
    int posicion=0;
    public char Id{get;set;}
    public MovidoEventHandler Movido;
    public void Mover()
    {
        posicion += Jugador.genAleatorio.Next(0,10);
        if (Movido != null)
        {
            Movido(this,new EventArgs());
        }
    }
    public void Imprimirse()
    {
        Console.WriteLine("Jugador {0} => posición {1}",Id,posicion);
    }
}
```

Puesto que no es necesario proveer información adicional, podría enviarse simplemente null. Sin embargo se recomienda instanciar y enviar un objeto EventArgs



# Ejemplo (Cont.)

```
using System;
class Programa
{
    static void Main()
    {
        Jugador j1=new Jugador() {Id='A'};
        Jugador j2=new Jugador() {Id='B'};
        j1.Movido=jugadorMovido;
        j2.Movido=jugadorMovido;
        for(int i=1;i<=10;i++)
        {
            j1.Mover();
            j2.Mover();
        }
        Console.ReadKey();
    }
    static void jugadorMovido(object sender, EventArgs e)
    {
        (sender as Jugador).Imprimirse();
    }
}
```

sender es de tipo object por lo tanto debe convertirse a Jugador para acceder a sus miembros.