



# .NET

## Teoría 8

# Delegados



# Delegados

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



Vamos a presentar el concepto por medio de un ejemplo



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria8`
4. Abrir `Visual Studio Code` sobre este proyecto



Codificar Auxiliar.cs y Program.cs de la siguiente manera y ejecutar



```
----- Auxiliar.cs -----
```

```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Console.WriteLine(SumaUno(10));
        Console.WriteLine(SumaDos(10));
    }
    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;
}
```

```
----- Program.cs -----
```

```
using Teoria8;

Auxiliar aux = new Auxiliar();
aux.Procesar();
```

## Delegados - Introducción

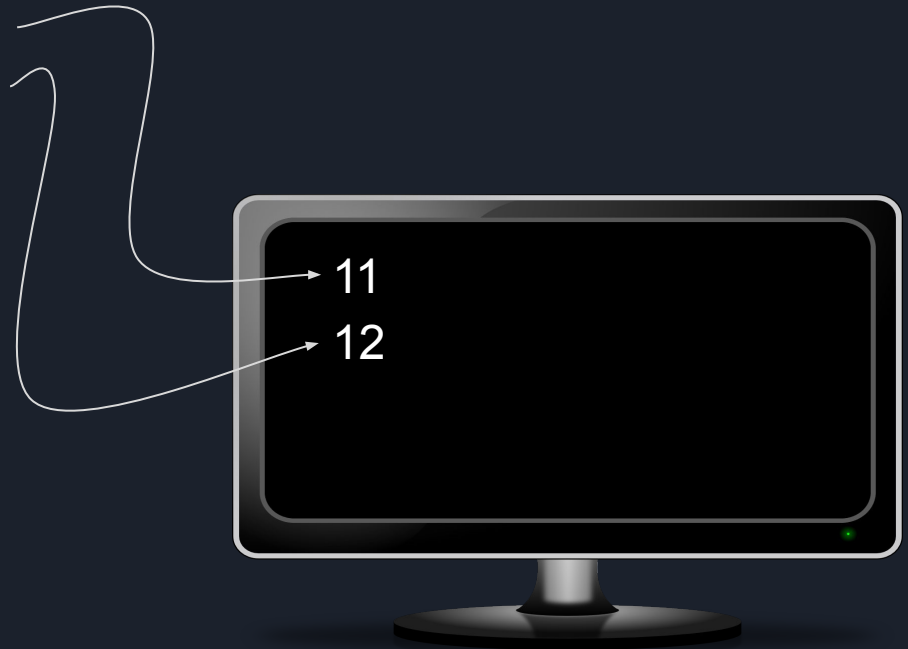
----- Auxiliar.cs -----

```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Console.WriteLine(SumaUno(10));
        Console.WriteLine(SumaDos(10));
    }
    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;
}
```

----- Program.cs -----

```
using Teoria8;

Auxiliar aux = new Auxiliar();
aux.Procesar();
```



# Asignación de métodos a variables

Queremos asignar métodos a variables

. . .

`f = SumaUno;`

`Console.WriteLine(f(10));`

`f = SumaDos;`

`Console.WriteLine(f(10));`

. . .

Y usar esas variables para invocarlos

¿De qué  
tipo debe  
ser f?



Tipo de variables que admiten métodos

Las variables que admiten  
métodos son de algún tipo

Delegado





# Definición de los tipos delegados

- Para definir un tipo de delegado, se usa una sintaxis similar a la definición de una **firma de método**. Solo hace falta agregar la palabra clave **delegate** a la definición. Ejemplo:

```
delegate int FuncionEntera(int n);
```

- El compilador genera una clase derivada de **System.Delegate** que coincide con la firma usada (en este caso, un método que devuelve un entero y tiene un argumento entero)

# Aclaración sobre la firma de un método

La documentación de Microsoft a veces resulta un poco confusa respecto del concepto de firma de un método en relación al tipo de retorno. Sin embargo en <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods> aclara:

Un tipo de retorno de un método no forma parte de la firma del método para fines de sobrecarga de métodos. Sin embargo, es parte de la firma del método al determinar la compatibilidad entre un delegado y el método al que apunta.





Codificar el delegado FuncionEntera (en el archivo FuncionEntera.cs) y modificar Auxiliar.cs



```
-----FuncionEntera.cs-----
```

```
namespace Teoria8;  
delegate int FuncionEntera(int n);
```

```
-----Auxiliar.cs-----
```

```
namespace Teoria8;  
class Auxiliar  
{  
    public void Procesar()  
    {  
        FuncionEntera f;  
        f = SumaUno;  
        Console.WriteLine(f(10));  
        f = SumaDos;  
        Console.WriteLine(f(10));  
    }  
    int SumaUno(int n) => n + 1;  
    int SumaDos(int n) => n + 2;  
}
```

# Delegados - Introducción

-----FuncionEntera.cs-----

```
namespace Teoria8;  
delegate int FuncionEntera(int n);
```

-----Auxiliar.cs-----

```
namespace Teoria8;  
class Auxiliar  
{  
    public void Procesar()  
    {  
        FuncionEntera f;  
        f = SumaUno;  
        Console.WriteLine(f(10));  
        f = SumaDos;  
        Console.WriteLine(f(10));  
    }  
    int SumaUno(int n) => n + 1;  
    int SumaDos(int n) => n + 2;  
}
```

Se invoca  
SumaUno por  
medio de f

Se invoca  
SumaDos por  
medio de f





## Usar el método Invoke de los delegados



```
-----FuncionEntera.cs-----
```

```
namespace Teoria8;  
delegate int FuncionEntera(int n);
```

```
-----Auxiliar.cs-----
```

```
namespace Teoria8;  
class Auxiliar  
{  
    public void Procesar()  
    {  
        FuncionEntera f;  
        f = SumaUno;  
        Console.WriteLine(f.Invoke(10));  
        f = SumaDos;  
        Console.WriteLine(f.Invoke(10));  
    }  
    int SumaUno(int n) => n + 1;  
    int SumaDos(int n) => n + 2;  
}
```

También se pueden invocar los métodos en los delegados de forma explícita utilizando el método `Invoke`

# Asignación de 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) ;
```





### Agregar los siguientes métodos en la clase Auxiliar



```
...  
void Aplicar(int[] v, FuncionEntera f)  
{  
    for (int i = 0; i < v.Length; i++)  
    {  
        v[i] = f(v[i]);  
    }  
}  
void Imprimir(int[] v)  
{  
    foreach (int i in v)  
    {  
        Console.Write(i + " ");  
    }  
    Console.WriteLine();  
}
```

Recibe como parámetros un vector y una función en un delegado

Aplica la función `f` a cada uno de los elementos del vector `v`



# Modificar el método Procesar de la clase Auxiliar y ejecutar

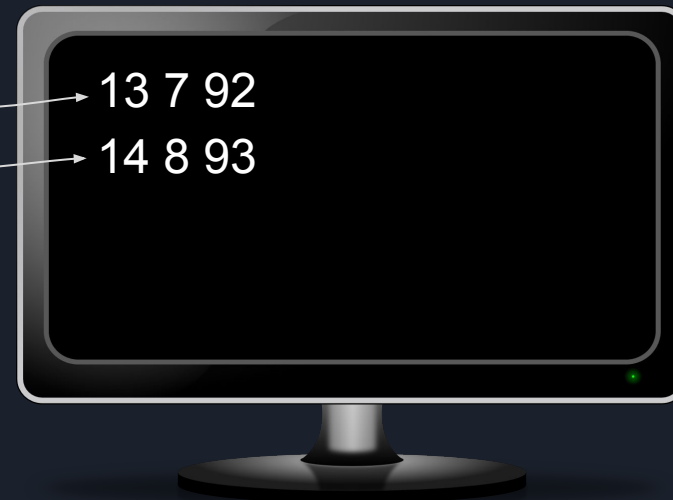


```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        int[] v = new int[] { 11, 5, 90 };
        Aplicar(v, SumaDos);
        Imprimir(v);
        Aplicar(v, SumaUno);
        Imprimir(v);
    }
    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;
    void Aplicar(int[] v, FuncionEntera f)
    {
        . . .
    }
}
```



# Delegados - Pasar métodos como parámetros

```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        int[] v = new int[] { 11, 5, 90 };
        Aplicar(v, SumaDos);
        Imprimir(v);
        Aplicar(v, SumaUno);
        Imprimir(v);
    }
    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;
    void Aplicar(int[] v, FuncionEntera f)
    {
        for (int i = 0; i < v.Length; i++)
        {
            v[i] = f(v[i]);
        }
    }
    void Imprimir(int[] v)
    {
        foreach (int i in v)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();
    }
}
```



# Métodos anónimos

- En ocasiones los métodos sólo se utilizan para crear una instancia de un delegado.
- Los métodos anónimos permiten prescindir del método con nombre definido por separado.
- Un método anónimo es un método que se declara en línea, en el momento de crear una instancia de un delegado

### Métodos anónimos - sintaxis

La sintaxis de un método anónimo incluye :

- La palabra clave `delegate`
- La `lista de parámetros` (si son necesarios)
- El `bloque de sentencias` con la implementación del método

```
delegate (parámetros) { implementación };
```



### Métodos anónimos - sintaxis

Observar que tiene la forma de un método cambiando su nombre por la palabra clave `delegate` y sin tipo de retorno

```
delegate (parámetros)  
{  
    implementación  
};
```

El tipo de retorno debe coincidir con el de la variable delegado a la que se asigne





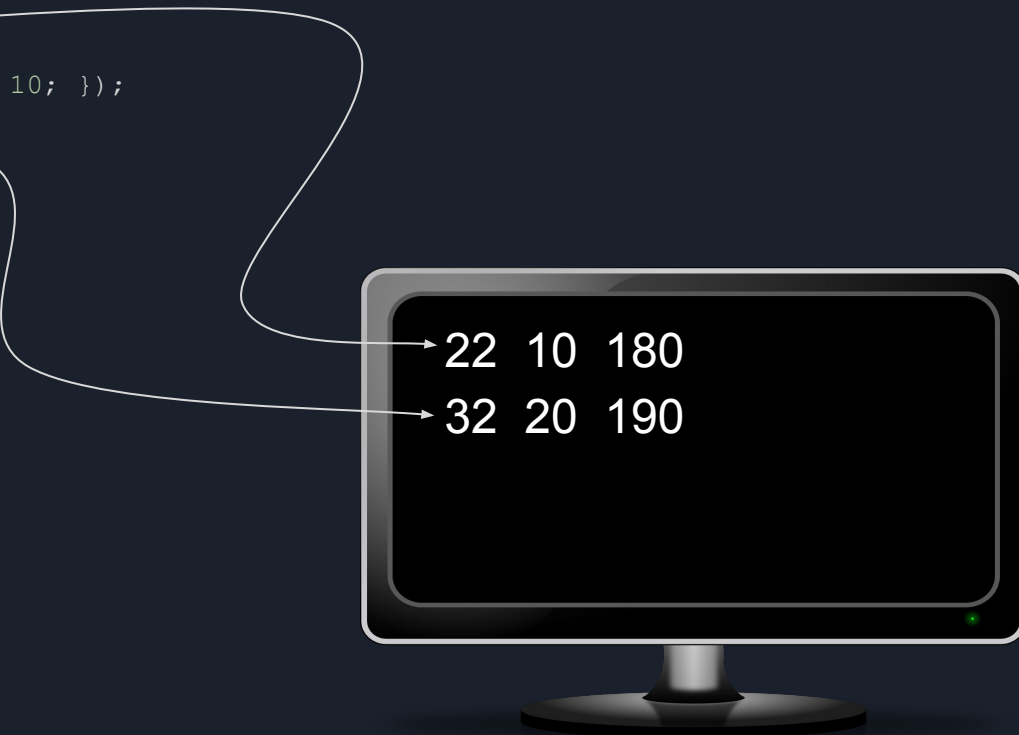
# Modificar Program.cs



```
class Auxiliar
{
    public void Procesar()
    {
        int[] v = new int[] { 11, 5, 90 };
        FuncionEntera f = delegate (int n)
        {
            return n * 2;
        };
        Aplicar(v, f);
        Imprimir(v);
        Aplicar(v, delegate (int n) { return n + 10; });
        Imprimir(v);
    }
    . . .
}
```

# Delegados - Métodos anónimos

```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        int[] v = new int[] { 11, 5, 90 };
        FuncionEntera f = delegate (int n)
        {
            return n * 2;
        };
        Aplicar(v, f);
        Imprimir(v);
        Aplicar(v, delegate (int n) { return n + 10; });
        Imprimir(v);
    }
    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;
    void Aplicar(int[] v, FuncionEntera f)
    {
        for (int i = 0; i < v.Length; i++)
        {
            v[i] = f(v[i]);
        }
    }
    void Imprimir(int[] v)
    {
        foreach (int i in v)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();
    }
}
```



22 10 180  
32 20 190

# Métodos anónimos

- Los **métodos anónimos** pueden acceder a sus **variables locales** y a las definidas en el entorno que lo rodea (**variables externas**).

```
int externa = 7;  
FuncionEntera f = delegate (int n)  
{  
    return n * 2 + externa;  
};  
Console.WriteLine(f(10));
```



# Expresiones lambda

- Los **métodos anónimos** se introdujeron en **C# 2.0** y las **expresiones lambda** en **C# 3.0** con el mismo propósito pero con sintaxis simplificada.
- Se puede transformar un método anónimo en una expresión lambda haciendo lo siguiente:
  - Eliminar la palabra clave **delegate**.
  - Colocar el operador lambda ( **=>** ) entre la lista de parámetros y el cuerpo del método anónimo.

```
f = delegate (int n) { return n * 2; };
```

```
f = (int n) => { return n * 2; };
```

Expresión  
lambda



## Expresiones lambda

Pero aún es posible otras simplificaciones sintácticas:

- Si no existen parámetros `ref`, `in` o `out`, el tipo de los parámetros puede omitirse:

```
f = (n) => { return n * 2; };
```

- Si hay un único parámetro, pueden omitirse los paréntesis:

```
f = n => { return n * 2; };
```

# Expresiones lambda

- Si el bloque de instrucciones es sólo una expresión de retorno, puede reemplazarse todo el bloque por la expresión de retorno:

```
f = n => n * 2;
```

- **Nota:** Si el delegado no tiene parámetros se deben usar paréntesis vacíos:

```
linea = () => Console.WriteLine();
```



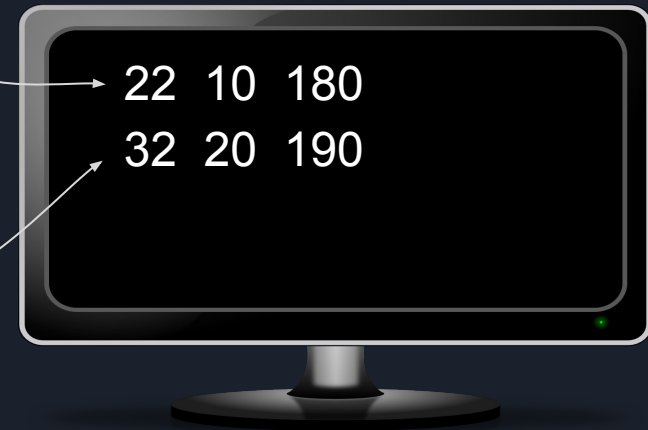
# Modificar el método Procesar de la clase Auxiliar usando expresiones lambda y ejecutar



```
class Auxiliar
{
    public void Procesar()
    {
        int[] v = new int[] { 11, 5, 90 };
        Aplicar(v, n => n * 2);
        Imprimir(v);
        Aplicar(v, n => n + 10);
        Imprimir(v);
    }
    . . .
}
```

# Delegados - Expresiones lambda

```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        int[] v = new int[] { 11, 5, 90 };
        Aplicar(v, n => n * 2);
        Imprimir(v);
        Aplicar(v, n => n + 10);
        Imprimir(v);
    }
    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;
    void Aplicar(int[] v, FuncionEntera f)
    {
        for (int i = 0; i < v.Length; i++)
        {
            v[i] = f(v[i]);
        }
    }
    void Imprimir(int[] v)
    {
        foreach (int i in v)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();
    }
}
```





# Modificar la clase Auxiliar de la siguiente manera y ejecutar



Encolando más delegados en a

```
class Auxiliar
{
    public void Procesar()
    {
        Action a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        a();
    }
    void Metodo1()
    => Console.WriteLine("Ejecutando Método1");
    void Metodo2()
    => Console.WriteLine("Ejecutando Método2");
    . . .
}
```

Action es un delegado ya definido en la BCL de la siguiente manera:

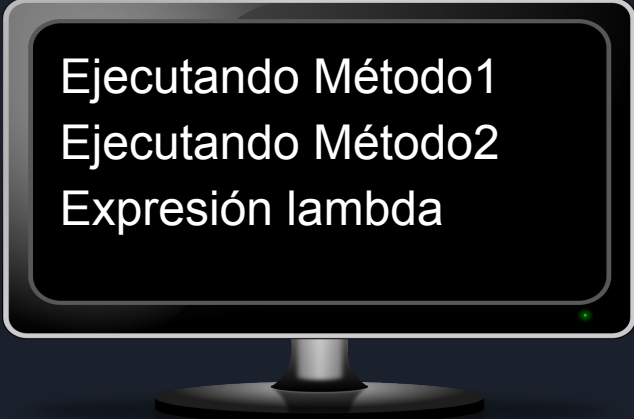
```
public delegate void Action();
```

# Delegados - Multidifusión

```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        a(); ←
    }
    void Metodo1()
        => Console.WriteLine("Ejecutando Método1");
    void Metodo2()
        => Console.WriteLine("Ejecutando Método2");
}
```

. . .

Un delegado puede llamar  
a más de un método  
cuando se invoca.  
Esto se conoce como  
**multidifusión**



Ejecutando Método1  
Ejecutando Método2  
Expresión lambda



# Modificar el método Procesar() de la clase Auxiliar



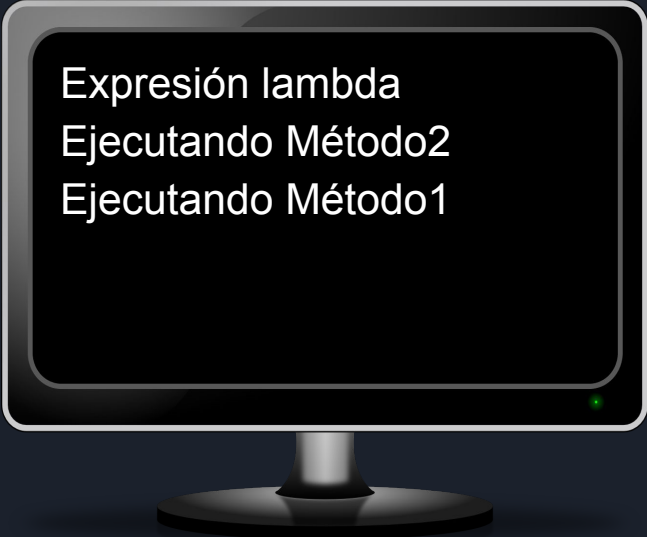
```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        Delegate[] encolados = a.GetInvocationList();
        for (int i = encolados.Length - 1; i >= 0; i--)
        {
            (encolados[i] as Action)?.Invoke();
        }
    }
}
```

# Delegados - Multidifusión

```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        Delegate[] encolados = a.GetInvocationList();
        for (int i = encolados.Length - 1; i >= 0; i--)
        {
            (encolados[i] as Action)?.Invoke();
        }
    }
    static void Metodo1()
        => Console.WriteLine("Ejecutando Método1");
    static void Metodo2()
        => Console.WriteLine("Ejecutando Método2");
    ...
}
```

Devuelve un arreglo de objetos Delegate, que corresponden la lista de delegados encolados

Invocando a los delegados en orden inverso



Expresión lambda  
Ejecutando Método2  
Ejecutando Método1





# Modificar el método Procesar() de la clase Auxiliar

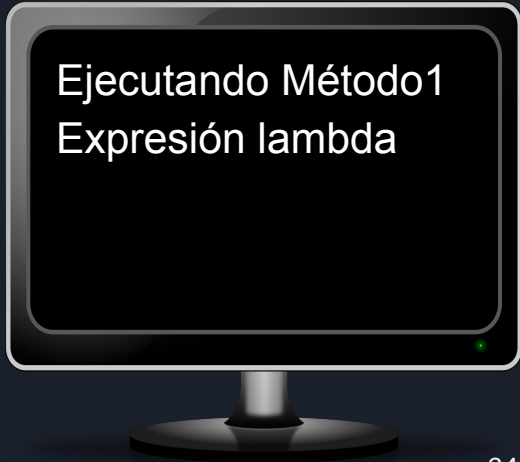


```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action? a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        a -= Metodo2;
        a?.Invoke();
    }
    . . .
}
```

Al quitar `Metodo2` de `a`, si fuese el único método encolado, `a` quedaría en `null`. Para evitar el warning del compilador se declara `a` de tipo `Action?`

Se quita al `Metodo2` (en realidad al delegado que encapsuló al `Metodo2`) de la lista de invocación

```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action? a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        a -= Metodo2;
        a?.Invoke();
    }
    static void Metodo1()
        => Console.WriteLine("Ejecutando Método1");
    static void Metodo2()
        => Console.WriteLine("Ejecutando Método2");
}
```



Ejecutando Método1  
Expresión lambda

## Algunos detalles

...

```
Action? a = null;
```

```
a = a + Metodo1;
```

```
a = a - Metodo2;
```

```
a = a - Metodo1;
```

...

No hay error, el resultado de `null + Metodo1` es `Metodo1`

No hay error al intentar quitar un elemento que no está en la lista de invocación

Al quitar el único elemento de la lista de invocación, `a` queda establecido en `null`



# Eventos

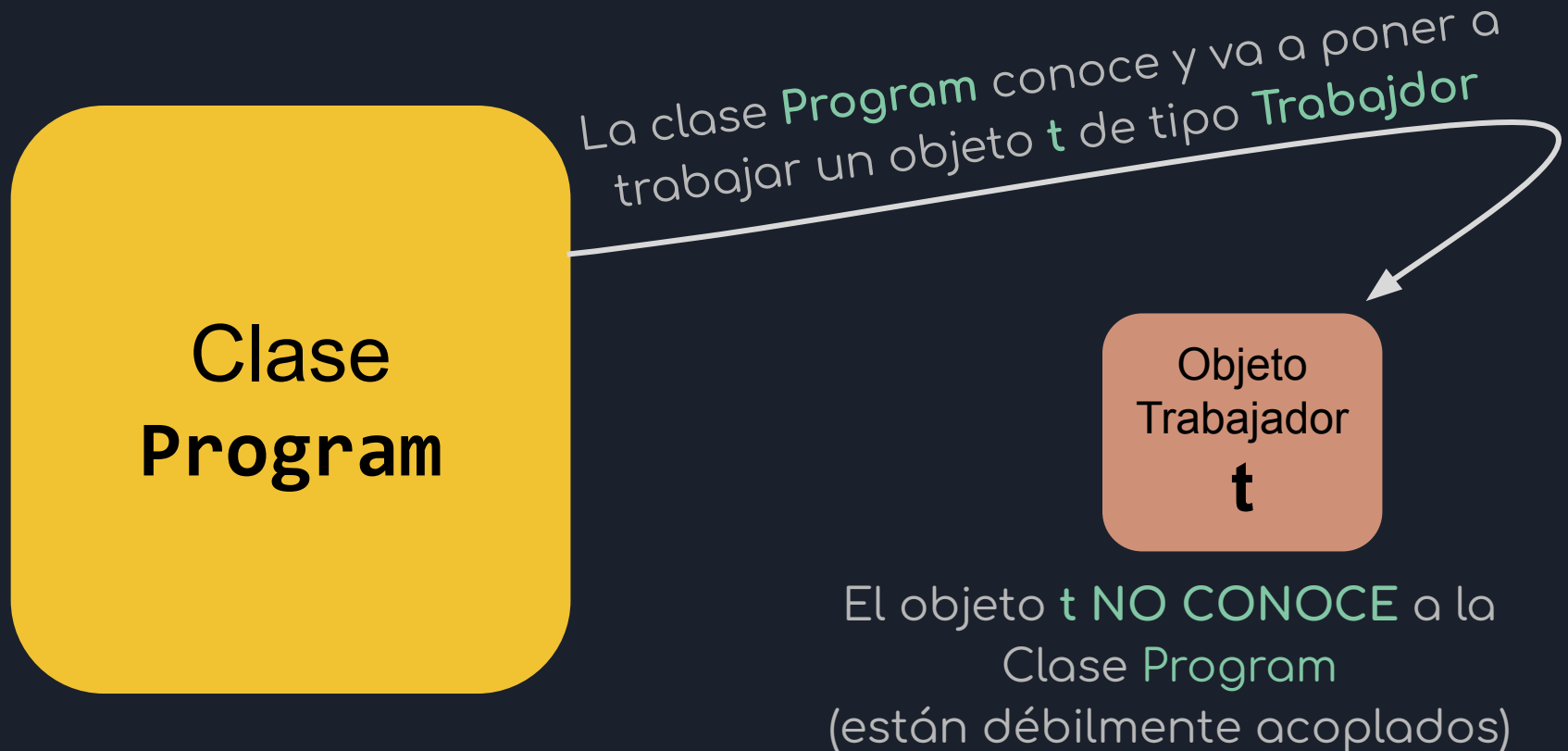
# Eventos

- Cuando **ocurre algo** importante, un objeto puede **notificar** el evento a otras **clases** u **objetos**.
- La clase que **produce** (o notifica) el evento recibe el nombre de **editor** y las clases que están interesadas en conocer la ocurrencia del evento se denominan **suscriptores**.
- Para que un suscriptor sea notificado, necesita estar **suscripto** al evento

# Características de los Eventos

- El **editor** determina **cuándo** se produce un evento; los **suscriptores** codifican en un método (manejador del evento) lo que harán cuando se produzca ese evento.
- Un **evento** puede tener **varios suscriptores**. Un **suscriptor** puede manejar **varios eventos** de **varios editores**.
- Nunca se provocan eventos que no tienen suscriptores.

### Eventos - Presentación de un caso de uso



**Program** se tiene que enterar cuanto **t** termina de trabajar  
pero queremos **mantener el bajo acoplamiento**

### Eventos - Presentación de un caso de uso





# Eventos - Implementación con delegados

-----Program.cs-----

```
using Teoria8;
```

```
Trabajador t = new Trabajador();
```

```
t.TrabajoFinalizado = ManejadorDelEvento;
```

```
t.Trabajar();
```

```
void ManejadorDelEvento()
```

```
=> Console.WriteLine("trabajo finalizado");
```

**TrabajoFinalizado** es un campo público de tipo delegado de **t**

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

-----Trabajador.cs-----

```
namespace Teoria8;
```

```
class Trabajador
```

```
{
```

```
    public Action? TrabajoFinalizado;
```

```
    public void Trabajar()
```

```
    {
```

```
        Console.WriteLine("trabajador trabajando...");
```

```
        // hace algún trabajo útil
```

```
        if (TrabajoFinalizado != null)
```

```
        {
```

```
            TrabajoFinalizado();
```

```
        }
```

```
    }
```

```
}
```

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**.  
Observar que **Trabajador** no conoce a quienes notifica

# Eventos - Implementación con delegados

```
-----Program.cs-----
```

```
using Teoria8;
```

```
Trabajador t = new Trabajador();
```

```
t.TrabajoFinalizado = ManejadorDelEvento;
```

```
t.Trabajar();
```

```
void ManejadorDelEvento()
```

```
=> Console.WriteLine("trabajo finalizado");
```

```
-----Trabajador.cs-----
```

```
namespace Teoria8;
```

```
class Trabajador
```

```
{
```

```
    public Action? TrabajoFinalizado;
```

```
    public void Trabajar()
```

```
    {
```

```
        Console.WriteLine("trabajador trabajando...");
```

```
        // hace algún trabajo útil
```

```
        if (TrabajoFinalizado != null)
```

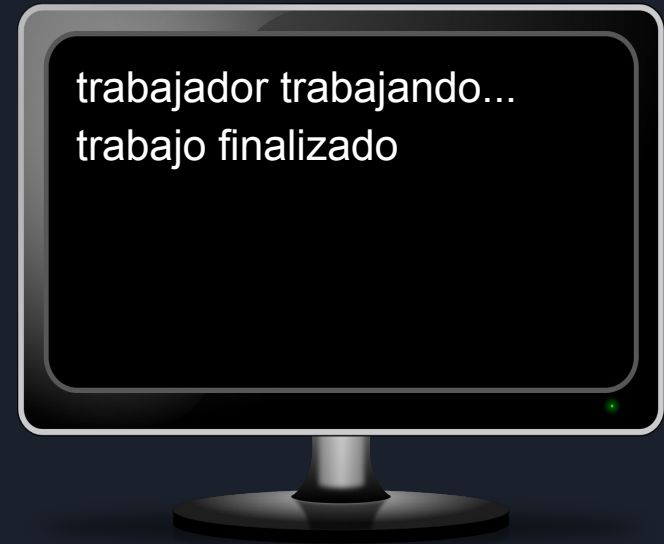
```
        {
```

```
            TrabajoFinalizado();
```

```
        }
```

```
    }
```

```
}
```



### Eventos - Convenciones

- Para los nombres de los eventos se recomiendan verbos en gerundio (ejemplo `IniciandoTrabajo`) o participio (ejemplo `TrabajoFinalizado`) según se produzcan antes o después del hecho de significación
- Los delegados usados para invocar a los manejadores de eventos deben tener 2 argumentos: uno de tipo `object` que contendrá al objeto que genera el evento y otro de tipo `EventArgs` (o derivado) para pasar argumentos. Además su tipo de retorno debe ser `void`

## El Tipo EventHandler

El tipo delegado **EventHandler** se utiliza para el caso de un evento que no requiere pasar datos como parámetros cuando se invoque el delegado

```
public delegate void EventHandler(object sender,  
                                   EventArgs e);
```

Es una clase vacía, no lleva datos,  
pero constituye la clase base de  
todas las que se utilizan para  
pasar argumentos



### Eventos - Convenciones

- Es deseable que los nombres que se utilicen compartan **una raíz común**.
- Por ejemplo, si define un evento **CapacidadExcedida**:
  - La clase para pasar los argumentos se debería denominar **CapacidadExcedidaEventArgs** y
  - el delegado asociado al evento, si no existe un tipo predefinido, se debería denominar **CapacidadExcedidaEventHandler**.

## Tipos predefinidos EventHandler

Más adelante en este curso, cuando veamos tipos genéricos presentaremos un conjunto de tipos predefinidos que hacen innecesario definir nuestros propios tipos delegados para los eventos



### Ejemplo de código 1

- Vamos a ver un ejemplo de codificación respetando las convenciones mencionadas
  - Se requiere codificar una clase `Trabajador`, con un método público `Trabajar` que produzca un evento `TrabajoFinalizado` una vez concluida su tarea.
  - Debe además comunicar (argumentos del evento) el `tiempo insumido` en la la ejecución del trabajo

### Ejemplo de código 1

Debido a que el evento que se lanza se llama **TrabajoFinalizado**, deberíamos definir los siguientes tipos:

```
class TrabajoFinalizadoEventArgs : EventArgs
{
    public TimeSpan TiempoConsumido { get; set; }
}
```

```
delegate void TrabajoFinalizadoEventHandler(
    object sender,
    TrabajoFinalizadoEventArgs e);
```



## Eventos - Implementación con delegados - Ejemplo respetando convenciones

-----TrabajoFinalizadoEventArgs.cs-----

```
class TrabajoFinalizadoEventArgs : EventArgs
{
    public TimeSpan TiempoConsumido { get; set; }
}
```

-----TrabajoFinalizadoEventHandler.cs-----

```
delegate void TrabajoFinalizadoEventHandler( object sender,
                                             TrabajoFinalizadoEventArgs e);
```

-----Trabajador.cs-----

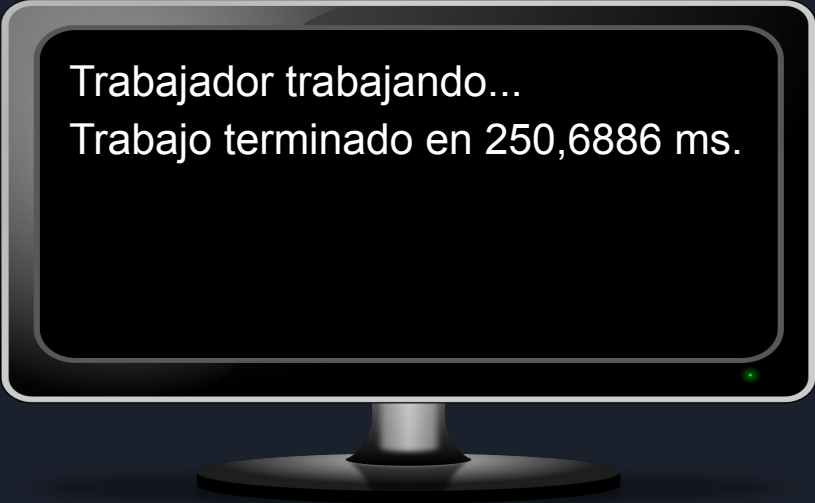
```
class Trabajador
{
    public TrabajoFinalizadoEventHandler? TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("Trabajador trabajando...");
        DateTime tInicial = DateTime.Now;
        for (int i = 1; i < 100_000_000; i++) ; //Pierdo tiempo simulando el trabajo
        TimeSpan lapso = DateTime.Now - tInicial;
        if (TrabajoFinalizado != null)
        {
            var e = new TrabajoFinalizadoEventArgs() { TiempoConsumido = lapso };
            TrabajoFinalizado(this, e);
        }
    }
}
```

## Eventos - Implementación con delegados - Ejemplo respetando convenciones

```
-----Program.cs-----
```

```
Trabajador t = new Trabajador();  
t.TrabajoFinalizado = t_TrabajoFinalizado;  
t.Trabajar();
```

```
void t_TrabajoFinalizado(object sender, TrabajoFinalizadoEventArgs e)  
{  
    string st = "Trabajo terminado en ";  
    st += $"{e.TiempoConsumido.TotalMilliseconds} ms.";  
    Console.WriteLine(st);  
}
```



Trabajador trabajando...  
Trabajo terminado en 250,6886 ms.

### Ejemplo de código 2

- Vamos a ver un ejemplo de codificación respetando las convenciones mencionadas en dónde se utiliza el objeto sender enviado cuando se lanza el evento
  - Se requiere una clase `Jugador`, con un método público `ArrojarDado` que produzca un evento `DadoArrojado` una vez obtenido el valor resultante.
  - Se instanciarán dos jugadores y se usará el mismo manejador para suscribirse al evento `DadoArrojado` de ambos
  - El programa finaliza cuando uno de ellos obtiene el número 6

## Eventos - Implementación con delegados - Otro ejemplo

-----Program.cs-----

```
bool seguirJugando = true;
```

```
Jugador j1 = new Jugador("Diana");
```

```
Jugador j2 = new Jugador("Pablo");
```

```
j1.DadoArrojado = DadoArrojado;
```

```
j2.DadoArrojado = DadoArrojado;
```

```
while (seguirJugando)
```

```
{
```

```
    j1.ArrojarDado();
```

```
    j2.ArrojarDado();
```

```
}
```

```
void DadoArrojado(object sender, DadoArrojadoEventArgs e)
```

```
{
```

```
    Console.WriteLine($"{(sender as Jugador)?.Nombre} -> {e.Valor}");
```

```
    if (e.Valor == 6)
```

```
    {
```

```
        seguirJugando = false;
```

```
    }
```

```
}
```



## Eventos - Implementación con delegados - Otro ejemplo

-----DadoArrojadoEventArgs.cs-----

```
class DadoArrojadoEventArgs : EventArgs {  
    public int Valor { get; set; }  
}
```

-----DadoArrojadoEventHandler.cs-----

```
delegate void DadoArrojadoEventHandler (object sender, DadoArrojadoEventArgs e);
```

-----Jugador.cs-----

```
class Jugador {  
    static Random s_random = new Random();  
    public string Nombre { get; }  
    public DadoArrojadoEventHandler? DadoArrojado;  
    public Jugador(string nombre) => Nombre = nombre;  
    public void ArrojarDado() {  
        int valor = s_random.Next(1, 7);  
        if (DadoArrojado != null) {  
            DadoArrojado(this, new DadoArrojadoEventArgs() { Valor = valor });  
        }  
    }  
}
```

## Observación 1

Observar que, gracias a la capacidad de **multidifusión** de los delegados, es posible que varias entidades se suscriban a un mismo evento, sólo tienen que conocer al que lo genera para encolar su propio manejador



## Observación 2

Cada uno de los suscriptores debería suscribirse al evento utilizando el operador `+=` para encolar su manejador sin eliminar los otros.

Pero no podemos garantizarlo porque dejamos público el campo delegado que representa al evento





## Event

- Un evento será un miembro definido con la palabra clave **Event**.
- Así como una **propiedad** controla el acceso a un campo de una clase u objeto, un evento lo hace con respecto a **campos** de tipo **delegados**, permitiendo ejecutar código cada vez que se añade o elimina un método del campo delegado.
- A diferencia de los delegados, a los eventos sólo se le pueden aplicar dos operaciones: **+=** y **-=**.



# Event

- Sintaxis

```
public event <TipoDelegado> NombreDelEvento
```

```
{
```

```
    add
```

```
    {
```

```
        <código add>
```

```
    }
```

```
    remove
```

```
    {
```

```
        <código remove>
```

```
    }
```

```
}
```

Código que se ejecutará cuando desde afuera se haga un **+=**  
En este bloque la variable implícita **value** contiene el delegado que se desea encolar

Código que se ejecutará cuando desde afuera se haga un **-=**  
En este bloque la variable implícita **value** contiene el delegado que se desea encolar

Es obligatorio codificar los dos descriptores (**add** y **remove**)

### Event

- Vamos a modificar la clase `Jugador` para que en lugar de publicar una variable de tipo delegado publique un evento.
- Vamos a establecer un control sobre este evento permitiendo sólo un suscriptor
- Comenzamos renombrando el campo `DadoArrojado` por `_dadoArrojado` y haciéndolo privado. Luego definimos el evento `DadoArrojado` que controlará el acceso al delegado

## Eventos - Definiendo un miembro Event

-----Jugador.cs-----

```
class Jugador {  
    static Random s_random = new Random();  
    public string Nombre { get; }  
    private DadoArrojadoEventHandler? _dadoArrojado;  
    public event DadoArrojadoEventHandler DadoArrojado {  
        add  
        {  
            if (_dadoArrojado == null) {  
                _dadoArrojado += value;  
            } else {  
                Console.WriteLine("Se denegó la suscripción");  
            }  
        }  
        remove  
        {  
            _dadoArrojado -= value;  
        }  
    }  
    public Jugador(string nombre) => Nombre = nombre;  
    public void ArrojarDado() {  
        int valor = s_random.Next(1, 7);  
        if (_dadoArrojado != null) {  
            _dadoArrojado(this, new DadoArrojadoEventArgs() { Valor = valor });  
        }  
    }  
}
```

Se renombró al  
hacerlo privado

Se encola sólo si no hay  
ninguno encolado

Evento

# Eventos - Definiendo un miembro Event

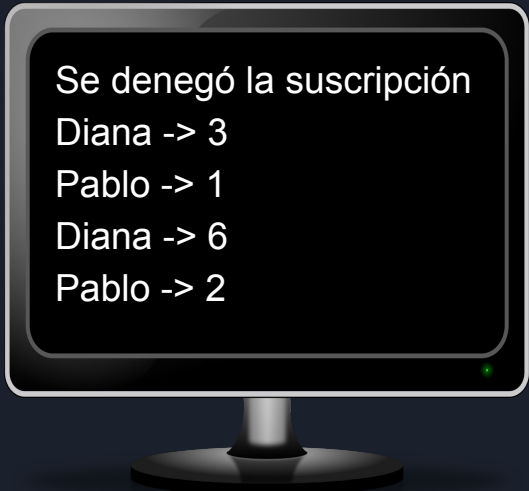
-----Program.cs-----

```
bool seguirJugando = true;
Jugador j1 = new Jugador("Diana");
Jugador j2 = new Jugador("Pablo");
j1.DadoArrojado += DadoArrojado;
j2.DadoArrojado += DadoArrojado;
j2.DadoArrojado += DadoArrojado;
while (seguirJugando)
{
    j1.ArrojarDado();
    j2.ArrojarDado();
}
```

Fue necesario  
cambiar = por += de  
lo contrario no  
compila

Intento de  
suscripción por  
segunda vez

```
void DadoArrojado(object sender, DadoArrojadoEventArgs e)
{
    Console.WriteLine($"{(sender as Jugador)?.Nombre} -> {e.Valor}");
    if (e.Valor == 6)
    {
        seguirJugando = false;
    }
}
```



Se denegó la suscripción  
Diana -> 3  
Pablo -> 1  
Diana -> 6  
Pablo -> 2

### Event - Notación abreviada

- En ocasiones no es necesario establecer control alguno en los descriptores de acceso `add` y `remove`
- Para estos casos `C#` provee una notación abreviada (similar a las propiedades automáticamente implementadas):

```
public event EventHandler TrabajoFinalizado;
```

- El compilador crea un campo privado de tipo `EventHandler` e implementa los descriptores de acceso `add` y `remove` para suscribirse y anular la suscripción al evento

Fin