



.NET

Teoría 8

Más sobre Interfaces Herencia


Interfaces - herencia

Las interfaces pueden heredar de múltiples interfaces

```
interface IInterface1 {  
    void Metodo1();  
}  
interface IInterface2 {  
    void Metodo2();  
}  
interface IInterface3: IInterface1, IInterface2 {  
    void Metodo3();  
}
```

```
class A : IInterface3 {  
    . . .  
}
```

La clase A debe
implementar Metodo1(),
Metodo2() y Metodo3()



Implementando múltiples Interfaces

```
interface IInterface1
{
    void Metodo1();
}
interface IInterface2
{
    void Metodo2();
}
```

```
class A : IInterface1, IInterface2
{
```

...

```
}
```

La clase A debe
implementar Metodo1() y
Metodo2()

Implementando Interfaces con miembros duplicados

```
interface IInterface1
{
    void Metodo();
}
interface IInterface2
{
    void Metodo();
}

class A : IInterface1, IInterface2
{
    public void Metodo()
    {
        . . .
    }
    . . .
}
```

Una única
implementación de
Metodo() implementa
las dos interfaces

Interrogante

Muy posiblemente los métodos
de igual nombre pero de
distintas interfaces, difieran
semánticamente.

¿Cómo implementarlos de
forma distinta ?



Respuesta



Implementación
explícita de
interfaces

Implementación explícita de miembros de interfaces

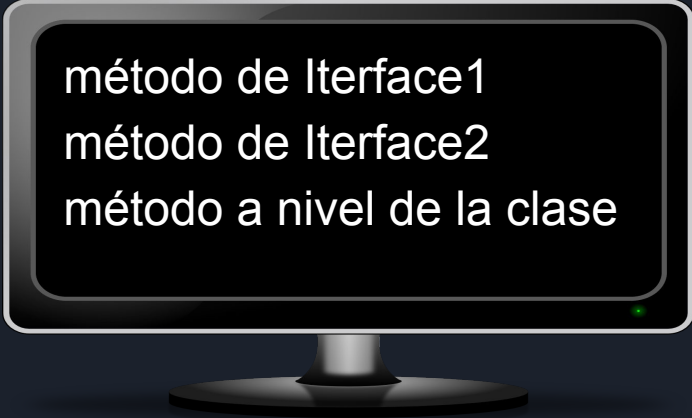
```
class A : IInterface1, IInterface2
{
    void IInterface1.Metodo() =>
        Console.WriteLine("método de Interface1");
    void IInterface2.Metodo() =>
        Console.WriteLine("método de Interface2");
    public void Metodo() =>
        Console.WriteLine("método a nivel de la clase");
}
```

IMPORTANTE:

La implementación explícita de un método de interface no lleva el modificador de acceso `public`

Implementación explícita de miembros de interfaces

```
. . .  
A objA = new A();  
(objA as IInterface1).Metodo();  
(objA as IInterface2).Metodo();  
objA.Metodo();  
. . .
```



método de IInterface1
método de IInterface2
método a nivel de la clase

Implementación explícita de miembros de interfaces

Cuando hay **implementaciones explícitas** de miembros de **interfaz**, la implementación a nivel de clase está permitida pero no es requerida.

Por lo tanto se tienen los siguientes 3 escenarios

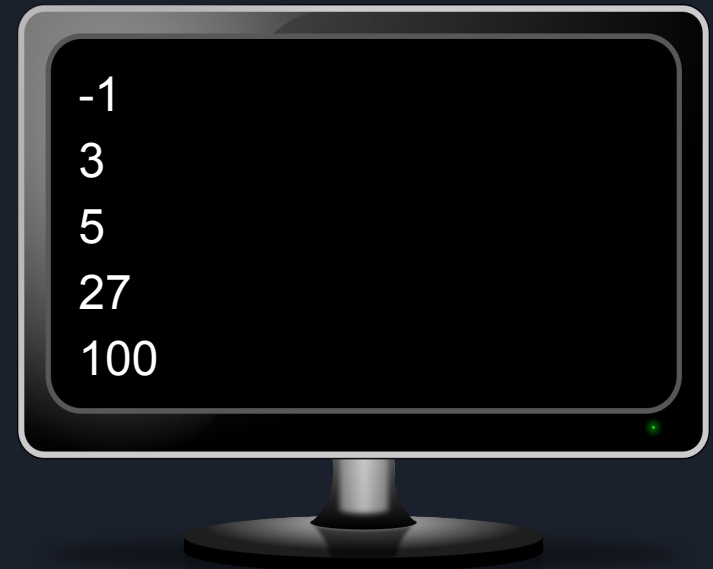
- una implementación a nivel de clase
- una implementación explícita de interface
- Ambas, una implementación explícita de interface y una implementación a nivel de clase

Interfaces de la plataforma que se usan para la comparación

Interface IComparable. Ejemplo de ordenamiento

```
var vector = new int[] { 27, 5, 100, -1, 3 };  
Array.Sort(vector);  
foreach (int i in vector)  
{  
    Console.WriteLine(i);  
}
```

Ordenar un vector es muy simple
utilizando el método estático `Sort`
de la clase `Array`



Interface IComparable. Ejemplo de ordenamiento

El método **Sort** de **Array** funciona correctamente porque todos los elementos del vector (en este caso de tipo **int**) son comparables entre sí porque implementan la interface **IComparable**





Crear la aplicación de consola Teoria8



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 la clase Empleado



```
namespace Teoria8;

class Empleado
{
    public string Nombre { get; private set; }
    public Empleado(string nombre)
    {
        Nombre = nombre;
    }
    public void Imprimir()
    => Console.WriteLine($"Soy el empleado {Nombre}");
}
```



Ordenamiento - Ejemplo 2

Codificar Program.cs de la siguiente manera



```
using Teoria8;

var vector = new Empleado[] {
    new Empleado("Juan"),
    new Empleado("Adriana"),
    new Empleado("Diego")
};

Array.Sort(vector);
foreach (Empleado e in vector)
{
    e.Imprimir();
}
```


Interfaces - System.IComparable

```
2
3 var vector = new Empleado[] {
4     new Empleado("Juan"),
5     new Empleado("Adriana"),
6     new Empleado("Diego")
7 };
8 Array.Sort(vector);
```

El método **Sort()** de **Array** provoca un error en tiempo de ejecución (Excepción) al intentar comparar dos elementos que no son comparables entre sí porque no implementan la interfaz **IComparable**

Exception has occurred: CLR/System.InvalidOperationException ×

Excepción no controlada del tipo 'System.InvalidOperationException' en System.Private.CoreLib.dll: 'Failed to compare two elements in the array.'

Se encontraron excepciones internas, consulte \$exception en la ventana de variables para obtener más detalles.

Excepción más interna System.ArgumentException : At least one object must implement IComparable,
en System.Collections.Comparer.Compare(Object a, Object b)
en System.Collections.Generic.ObjectComparer`1.Compare(T x, T y)
en System.Collections.Generic.ArraySortHelper`1.SwapIfGreater(Span`1 keys, Comparison`1 comparer, Int32 i, Int32 j)
en System.Collections.Generic.ArraySortHelper`1.IntroSort(Span`1 keys, Int32 depthLimit, Comparison`1 comparer)
en System.Collections.Generic.ArraySortHelper`1.IntrospectiveSort(Span`1 keys, Comparison`1 comparer)
en System.Collections.Generic.ArraySortHelper`1.Sort(Span`1 keys, IComparer`1 comparer)

Interface IComparable

¿ Se acuerdan del polimorfismo,
`Console.WriteLine()` y `ToString()` ?

Aunque no podemos modificar el método
`Sort()` de `Array` podemos hacer que
funcione con nuestras clases enseñando a
los objetos de estas clases a compararse
entre sí implementando la interfaz
`IComparable`



Interface IComparable

```
namespace System
{
    // Summary:
    //     Defines a generalized type-specific comparison method that a value type or class
    //     implements to order or sort its instances.
    public interface IComparable
    {
        //     Compares the current instance with another object of the same type and returns
        //     an integer that indicates whether the current instance precedes, follows, or
        //     occurs in the same position in the sort order as the other object.
        int CompareTo(object? obj);
    }
}
```

Valores de retorno del método CompareTo

(< 0) si this está antes que obj

(= 0) si this ocupa la misma posición que obj

(> 0) si this está después que obj



Solución ordenamiento - Ejemplo 2

Implementar la interfaz IComparable



```
class Empleado : IComparable
{
    public int CompareTo(object? obj)
    {
        int result = 0;
        if (obj is Empleado)
        {
            string nombre = ((Empleado)obj).Nombre;
            result = this.Nombre.CompareTo(nombre);
        }
        return result;
    }
}
```

. . .

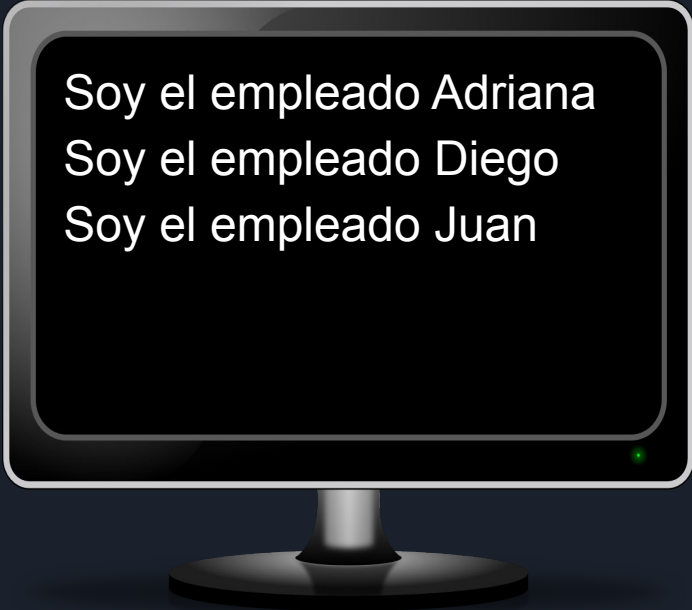
Ordenamiento Ejemplo 2



```
using Teoria8;

var vector = new Empleado[] {
    new Empleado("Juan"),
    new Empleado("Adriana"),
    new Empleado("Diego")
};

Array.Sort(vector);
foreach (Empleado e in vector)
{
    e.Imprimir();
}
```



Soy el empleado Adriana
Soy el empleado Diego
Soy el empleado Juan

Interface IComparer Ejemplo de ordenamiento

Si queremos otro criterio de orden, podemos utilizar una sobrecarga del método `Array.Sort()` que recibe también como argumento un objeto comparador que debe implementar la interfaz `IComparer`



Interface IComparer

```
namespace System.Collections
{
    //
    // Summary:
    //     Exposes a method that compares two objects.
    public interface IComparer
    {
        //
        // Summary:
        //     Compares two objects and returns a value indicating whether one is less than,
        //     equal to, or greater than the other.
        //
        // Returns:
        //     A signed integer that indicates the relative values of x and y:
        //     - If less than 0, x is less than y.
        //     - If 0, x equals y.
        //     - If greater than 0, x is greater than y.

        int Compare(object? x, object? y);
    }
}
```



Ordenamiento Ejemplo 3

Para este ejemplo modificamos la clase Empleado

```
class Empleado : IComparable
```

```
{
```

```
    public int Legajo { get; set; }
```

← Agregar la
propiedad
Legajo

```
    public void Imprimir()
```

```
    {
```

```
        Console.Write($"Soy el empleado {Nombre}");
```

```
        Console.WriteLine($"  Legajo: {Legajo}" );
```

```
    }
```

```
    . . .
```

Modificamos el método
Imprimir() de la clase
Empleado

```
}
```


Ordenamiento Ejemplo 3

```
namespace Teoria8;

class ComparadorPorLegajo : System.Collections.IComparer
{
    public int Compare(object? x, object? y)
    {
        int result = 1;
        if (x is Empleado && y is Empleado)
        {
            int legajo1 = ((Empleado)x).Legajo;
            int legajo2 = ((Empleado)y).Legajo;
            result = legajo1.CompareTo(legajo2);
        }
        return result;
    }
}
```

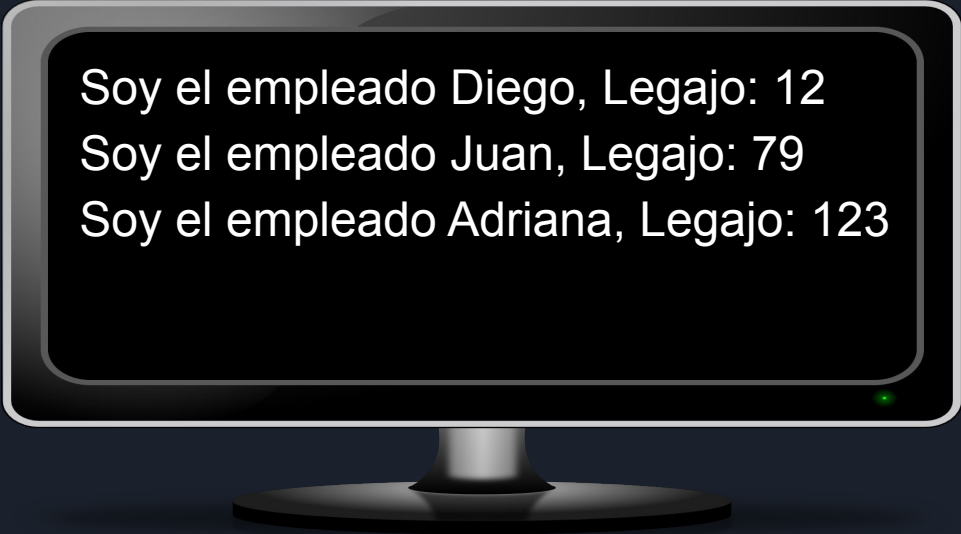
Definimos una nueva clase especializada en comparar empleados por algún criterio. Esta clase va a implementar la interfaz `IComparer`

```
using Teoria8;
```

```
var vector = new Empleado[] {  
    new Empleado("Juan") {Legajo=79},  
    new Empleado("Adriana") {Legajo=123},  
    new Empleado("Diego") {Legajo=12}  
};
```

```
Array.Sort(vector, new ComparadorPorLegajo());  
foreach (Empleado e in vector)  
{  
    e.Imprimir();  
}
```

Ordenamiento
por legajo



Soy el empleado Diego, Legajo: 12
Soy el empleado Juan, Legajo: 79
Soy el empleado Adriana, Legajo: 123

Ordenamiento - Ejemplo 4

```
class ComparadorPorLegajo : System.Collections.IComparer
{
    public bool Descendente { get; set; } = false;

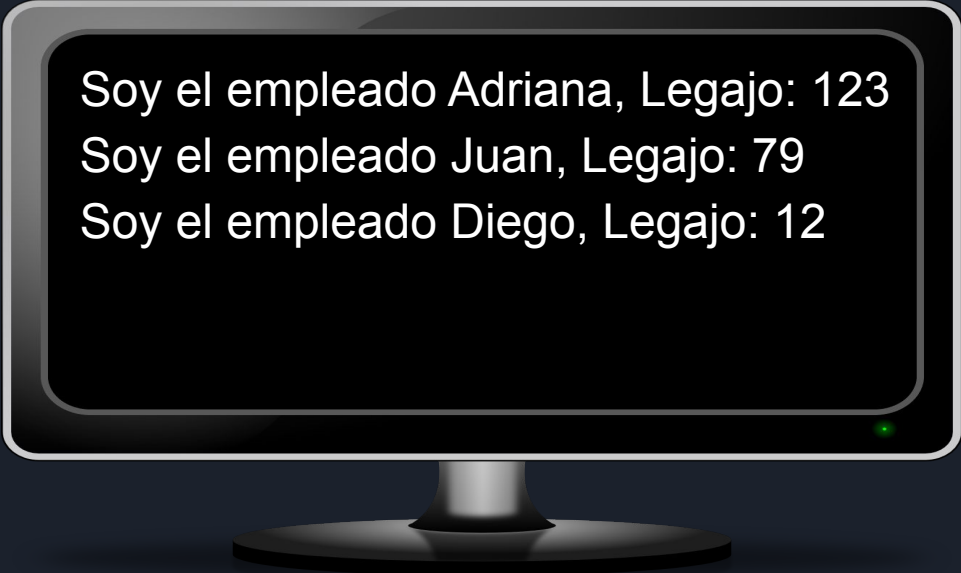
    public int Compare(object? x, object? y)
    {
        int result = 1;
        if (x is Empleado && y is Empleado)
        {
            int legajo1 = ((Empleado)x).Legajo;
            int legajo2 = ((Empleado)y).Legajo;
            result = legajo1.CompareTo(legajo2);
        }
        if (Descendente)
        {
            result = -result;
        }
        return result;
    }
}
```

Modificando
`ComparadorPorLegajo` para
permitir ordenar ascendente o
descendentemente

```
using Teoria8;

var vector = new Empleado[] {
    new Empleado("Juan") {Legajo=79},
    new Empleado("Adriana") {Legajo=123},
    new Empleado("Diego") {Legajo=12}
};

Array.Sort(vector, new ComparadorPorLegajo() { Descendente = true });
foreach (Empleado e in vector)
{
    e.Imprimir();
}
```



Soy el empleado Adriana, Legajo: 123
Soy el empleado Juan, Legajo: 79
Soy el empleado Diego, Legajo: 12

Interfaces de la plataforma que se
utilizan para “enumerar”

`System.Collections.IEnumerable`

y

`System.Collections.IEnumerator`

Uso de la instrucción foreach Ejemplo 1

```
. . .  
string[] vector = new string[] {"uno", "dos", "tres"};  
foreach(string st in vector)  
{  
    Console.WriteLine(st);  
}  
. . .
```

`vector` es un objeto enumerable, por eso puede usarse con la instrucción `foreach`





Codificar la clase Pyme



```
namespace Teoria8;

class Pyme
{
    Empleado[] empleados = new Empleado[3];
    public Pyme(Empleado e1, Empleado e2, Empleado e3)
    {
        empleados[0] = e1;
        empleados[1] = e2;
        empleados[2] = e3;
    }
}
```



Codificar Program.cs de la siguiente manera e intentar compilar



```
using Teoria8;

Pyme miPyme = new Pyme(new Empleado("Juan"),
                        new Empleado("Adriana"),
                        new Empleado("Diego"));

foreach (Empleado e in miPyme)
{
    e.Imprimir();
}
```


Error de compilación

```
using Teoria8;
```

```
Pyme miPyme = new Pyme(new Empleado("Juan"),  
                        new Empleado("Adriana"),  
                        new Empleado("Diego"));
```

```
foreach (Empleado e in miPyme)  
{  
    e.Imprimir();  
}
```

Error de compilación:
'Pyme' no contiene ninguna definición de
extensión o instancia pública para
'GetEnumerator'
miPyme no es un objeto enumerable

Interface System.Collections.IEnumerable

Un tipo es enumerable si
implementa la interface
System.Collections.IEnumerable



Interface System.Collections.IEnumerable

```
namespace System.Collections
{
    public interface IEnumerable
    {
        // Returns an enumerator that
        // iterates through a collection.
        IEnumerator GetEnumerator();
    }
}
```

Observar que el método `GetEnumerator()` devuelve un objeto de tipo interface, es decir de algún tipo que implemente la interfaz `System.Collections.IEnumerator`



Modificar la clase Pyme para implementar la interfaz System.Collections.IEnumerable



```
using System.Collections;  
namespace Teoria8;
```

```
class Pyme: IEnumerable  
{  
    Empleado[] empleados = new Empleado[3];  
    public Pyme(Empleado e1, Empleado e2, Empleado e3)  
    {  
        empleados[0] = e1;  
        empleados[1] = e2;  
        empleados[2] = e3;  
    }  
}
```

Los arreglos implementan la interface `IEnumerable`, estamos aprovechando el enumerador que proveen

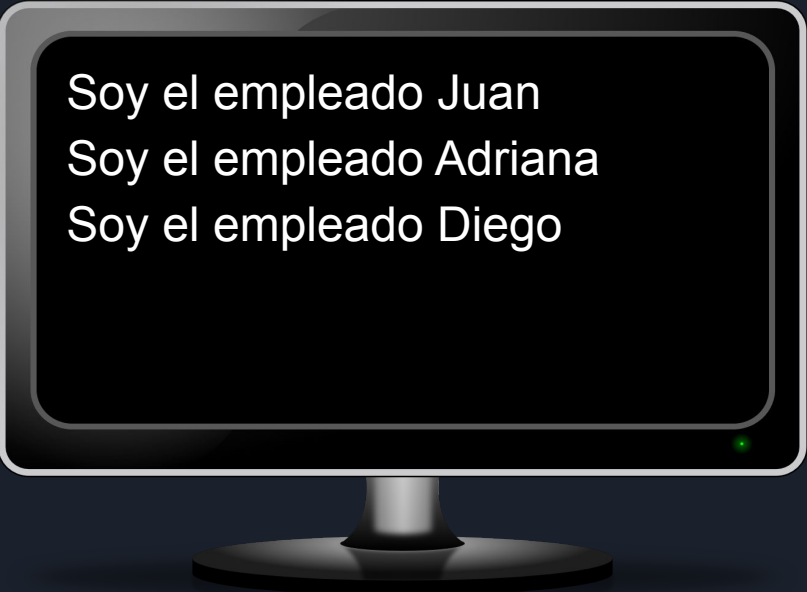
```
    public IEnumerator GetEnumerator()  
    {  
        return empleados.GetEnumerator();  
    }  
}
```

```
using Teoria8;

Pyme miPyme = new Pyme(new Empleado("Juan"),
                        new Empleado("Adriana"),
                        new Empleado("Diego"));

foreach (Empleado e in miPyme)
{
    e.Imprimir();
}
```

Solucionado !



Soy el empleado Juan
Soy el empleado Adriana
Soy el empleado Diego

¿ Qué es un enumerador ?

- Es un objeto que puede devolver los elementos de una colección, uno por uno, en orden, según se solicite.
- Un enumerador "conoce" el orden de los elementos y realiza un seguimiento de dónde está en la secuencia. Luego devuelve el elemento actual cuando se solicita.
- Un enumerador debe implementar la interface `System.Collection.IEnumerator`

Interface System.Collections.IEnumerator

```
namespace System.Collections
{
    public interface IEnumerator
    {
        // Gets the current element in the current position.
        object Current { get; }

        // Advances the enumerator to the next element
        // Returns true if the enumerator was successfully advanced
        bool MoveNext();

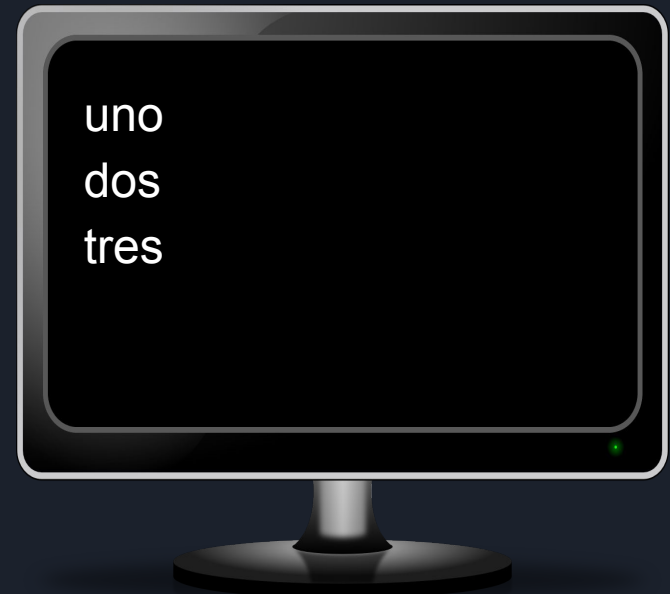
        // Sets the enumerator before the first element
        void Reset();
    }
}
```

Recorriendo un enumerador

```
using System.Collections;

var vector = new string[] {"uno", "dos", "tres"};
IEnumerator e = vector.GetEnumerator();

while (e.MoveNext())
{
    Console.WriteLine(e.Current);
}
```



Recorriendo un enumerador

```
using System.Collections;

var vector = new string[] {"uno", "dos", "tres"};
IEnumerator e = vector.GetEnumerator();

←
while (e.MoveNext())
{
    Console.WriteLine(e.Current);
}
```

Invocar aquí `e.Current` provocaría una excepción `InvalidOperationException`. Lo mismo ocurriría después de `e.Reset()`

Tip: Sólo invocar `e.Current` luego de obtener true con `e.MoveNext()`

Recorriendo un enumerador

```
using System.Collections;

var vector = new string[] {"uno", "dos", "tres"};
IEnumerator e = vector.GetEnumerator();

while (e.MoveNext())
{
    Console.WriteLine(e.Current);
}
```

Invocar aquí `e.Current` provocaría una excepción `InvalidOperationException`, porque la última ejecución de `e.MoveNext()` retornó false

Codificando un enumerador Ejemplo

Se requiere codificar una clase que implemente la interfaz `System.Collections.IEnumerator` para enumerar los nombres de las estaciones del año comenzando por “verano”

Interfaces - System.Collection.IEnumerator

```
using System.Collections;

class EnumeradorEstaciones : IEnumerator
{
    private string actual = "Inicio";

    public void Reset() => actual = "Inicio";

    public object Current =>
        (actual == "Inicio" || actual == "Fin") ? throw new InvalidOperationException() : actual;

    public bool MoveNext()
    {
        switch (actual)
        {
            case "Inicio": actual = "Verano"; break;
            case "Verano": actual = "Otoño"; break;
            case "Otoño": actual = "Invierno"; break;
            case "Invierno": actual = "Primavera"; break;
            case "Primavera": actual = "Fin"; break;
        }
        return (actual != "Fin");
    }
}
```

```
using System.Collections;

IEnumerator e = new EnumeradorEstaciones();
while (e.MoveNext())
{
    Console.WriteLine(e.Current);
}
```



Codificando un enumerable para usar con foreach. Ejemplo

```
using System.Collections;

class Estaciones : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        return new EnumeradorEstaciones();
    }
}
```

```
Estaciones estaciones = new Estaciones();  
foreach (string st in estaciones)  
{  
    Console.WriteLine(st);  
}
```



Nota

En realidad la sentencia `foreach` no necesita que la colección implemente la interfaz `IEnumerable`, sin embargo exige que exista un método con el nombre `GetEnumerator()` que devuelva un objeto que implemente la interfaz `IEnumerator`.



Iteradores

- Los **iteradores** constituyen una forma mucho más simple de crear **enumeradores** y **enumerables** (el compilador lo hace por nosotros).
- Utilizan la sentencia **yield**
 - **yield return**: devuelve un elemento de una colección y mueve la posición al siguiente elemento.
 - **yield break**: detiene la iteración.

Iteradores

- Un **bloque iterador** es un bloque de código que contiene una o más sentencias **yield**.
- Un **bloque iterador** puede contener múltiples sentencias **yield return** o **yield break** pero no se permiten sentencias **return**
- El tipo de retorno de un **bloque iterador** debe declararse **IEnumerator** o **IEnumerable**

Iteradores - ejemplo 1

```
using System.Collections;
```

```
IEnumerator enumerador = colores();
```

```
while (enumerador.MoveNext())
```

```
{
```

```
    Console.WriteLine(enumerador.Current);
```

```
}
```

Current es de tipo
object

```
IEnumerator colores()
```

```
{
```

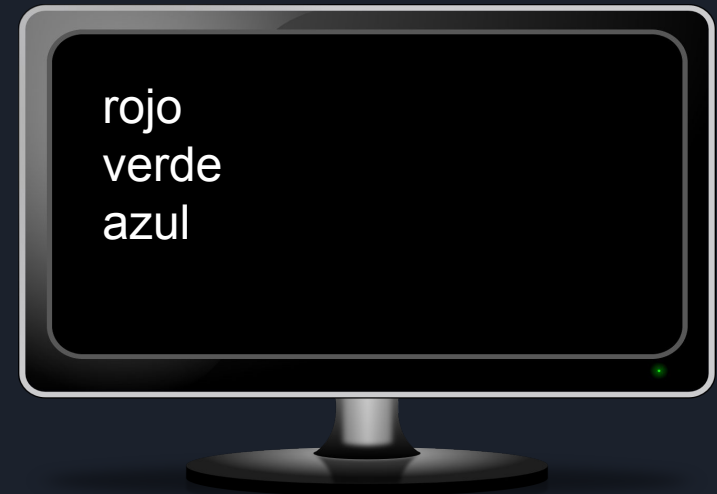
```
    yield return "rojo";
```

```
    yield return "verde";
```

```
    yield return "azul";
```

```
}
```

Este método es
un iterador



```
3 using System.Collections;
4
5 IEnumerator enumerador = colores();
6 while (enumerador.MoveNext())
7 {
8     Console.WriteLine(enumerador.Current);
9 }
10 enumerador.Reset();
```

Exception has occurred: CLR/System.NotSupportedException ✕

Excepción no controlada del tipo 'System.NotSupportedException' en Teoria7.dll: 'Specified method is not supported.'

en Program.<<<Main>\$>g__colores|0_0>d.System.Collections.IEnumerator.Reset()

en Program.<Main>\$(String[] args) en /home/leo/proyectos60/Teoria7/Program.cs: línea 10

```
11
12 IEnumerator colores()
13 {
14     yield return "rojo";
15     yield return "verde";
16     yield return "azul";
17 }
18
```



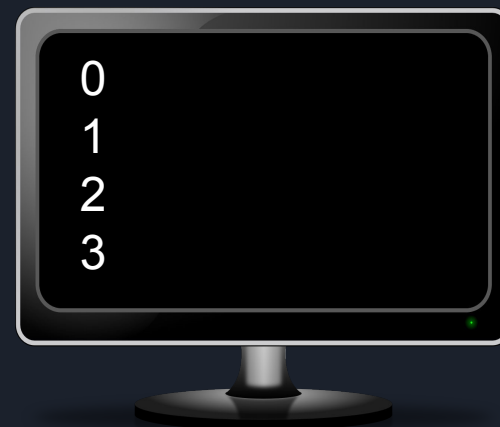
Cuidado!
Un enumerador
generado con un
iterador no
implementa el
método **Reset()**

Iteradores - ejemplo 2

```
using System.Collections;

IEnumerator e = Numeros();
while (e.MoveNext())
{
    Console.WriteLine(e.Current);
}

IEnumerator Numeros()
{
    int i = 0;
    while (true)
    {
        if (i <= 3) yield return i++;
        else yield break;
    }
}
```



IEnumerable generado por iterador

```
using System.Collections;

IEnumerable poderes = PoderesEstado();
foreach (var p in poderes)
{
    Console.WriteLine(p);
}

IEnumerable PoderesEstado()
{
    yield return "Ejecutivo";
    yield return "Legislativo";
    yield return "Judicial";
}
```

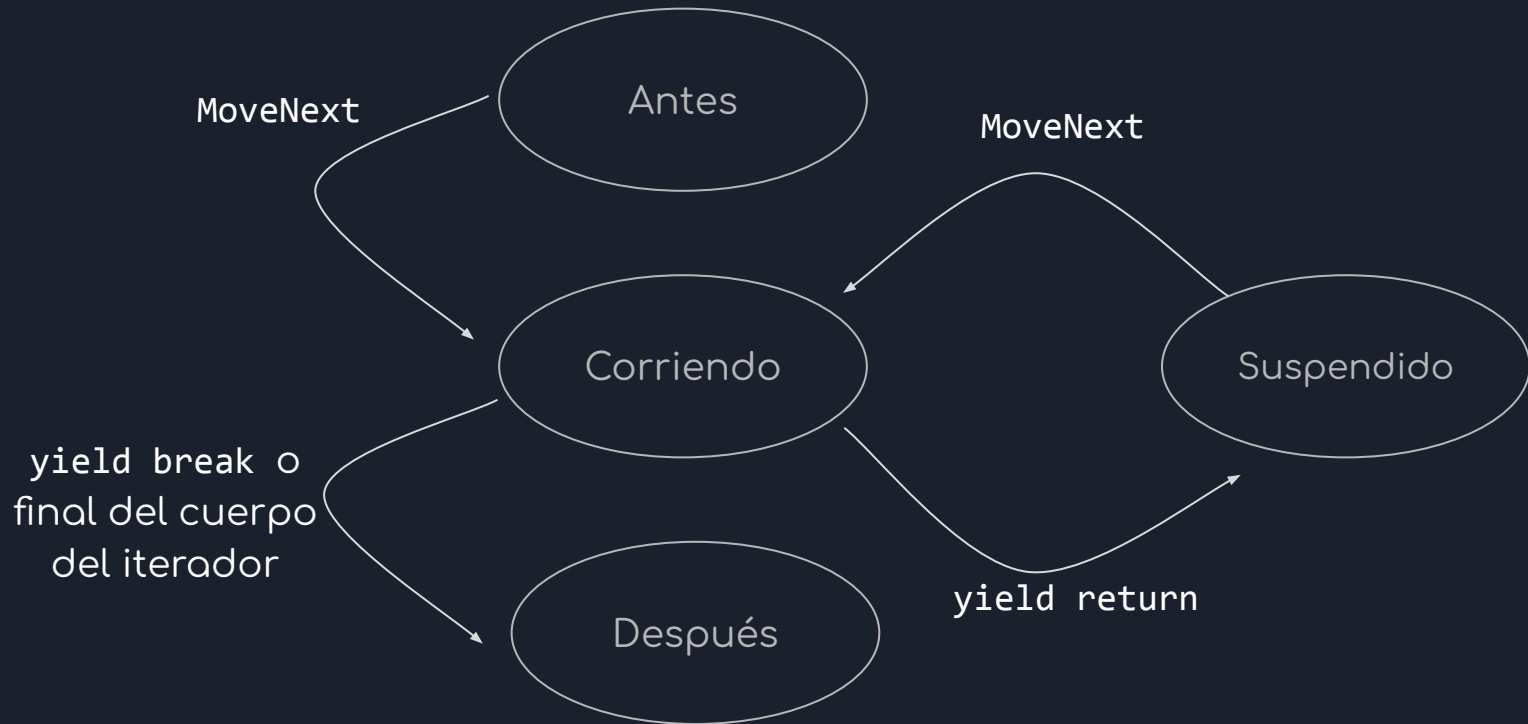
Alcanza con especificar
que el iterador devuelve
un `IEnumerable`.

¡ El compilador hace
todo el trabajo !



El detrás de escena de los iteradores

El enumerador generado por el compilador a partir de un iterador es una clase que implementa una máquina de estados



El detrás de escena de los iteradores

Un iterador produce un **enumerador**,
y **no una lista de elementos**. Este
enumerador es invocado por la
instrucción **foreach**. Esto permite
iterar a través de grandes cantidades
de datos sin leer todos los datos en
la memoria de una vez.



Interfaces - Iteradores

```
using System.Collections;

foreach (string st in GetA())
{
    Console.WriteLine(st);
    if (st == "AAAA")
    {
        break;
    }
}
```

```
IEnumerable GetA()
{
    string st = "";
    for (int i = 1; i < 1_000_000_000; i++)
    {
        yield return st += "A";
    }
}
```



El iterador no es un método que se va a ejecutar desde la primera a la última instrucción

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**



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();
```

Código en el archivo
08_RecursosParaLaTeoria.txt

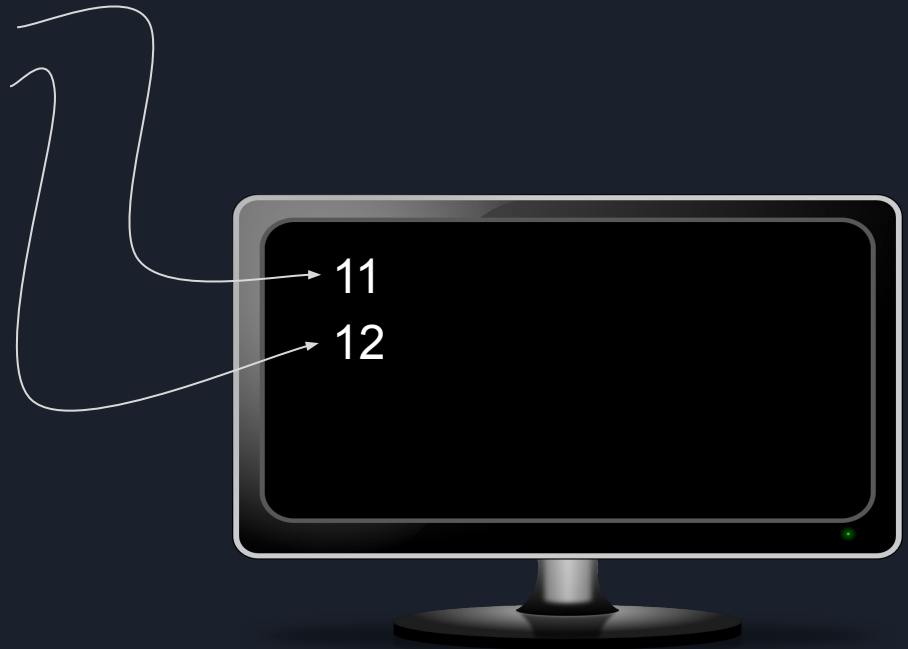
----- Auxiliar.cs -----

```
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 siguiente 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*



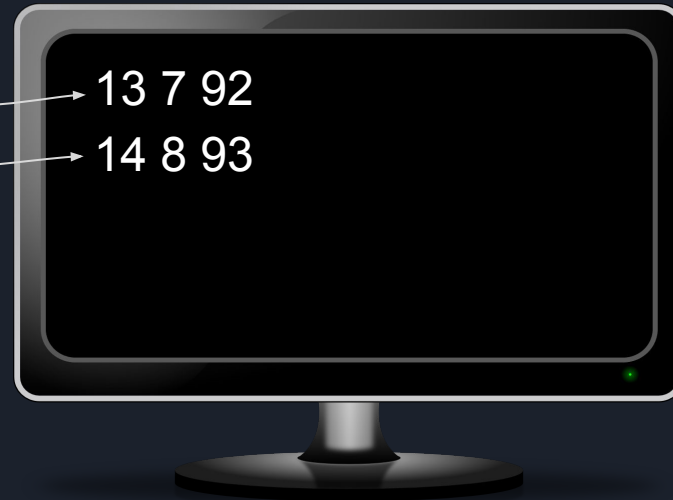
Modificar el método Procesar de la clase Auxiliar y ejecutar



```
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 se definen métodos con la intención de ser invocados sólo por medio de una variable de tipo 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();
```



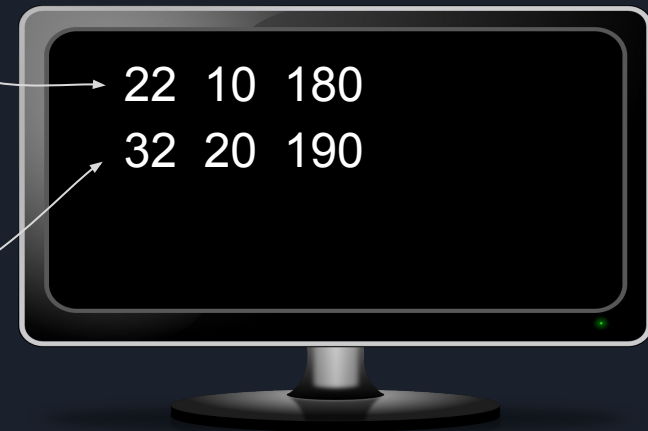
Modificar el método Procesar de la clase Auxiliar usando expresiones lambda y ejecutar



```
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();
    }
}
```



Fin de la teoría 8

Material complementario

El siguiente material es para quienes deseen conocer algo más sobre delegados y su utilización para implementar eventos.

Este contenido es opcional, no será evaluado en este curso



Modificar la clase Auxiliar de la siguiente manera y ejecutar



```
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");
```

```
    . . .
```

Encolando
más
delegados
en a

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

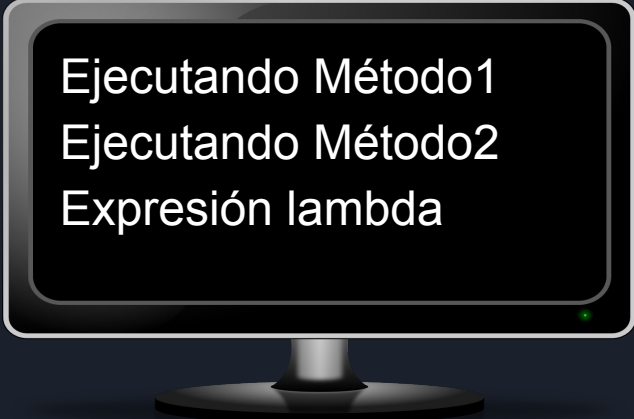
```
public delegate void Action();
```

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



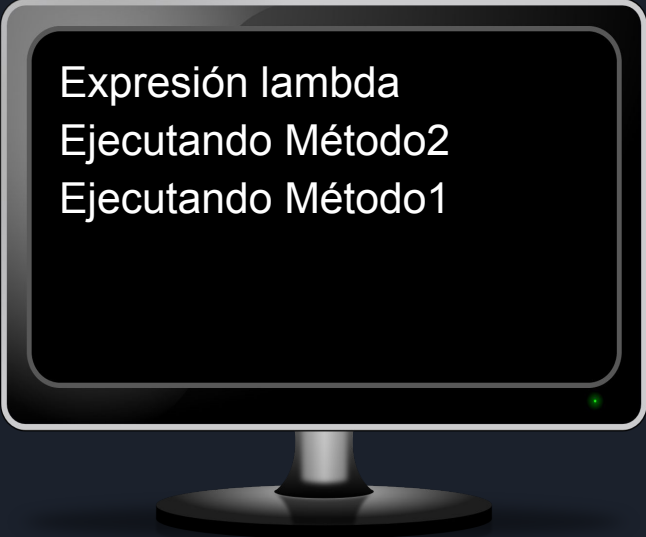
```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        Delegate[] encolados = a.GetInvocationList();
        for (int i = encolados.Length - 1; i >= 0; i--)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

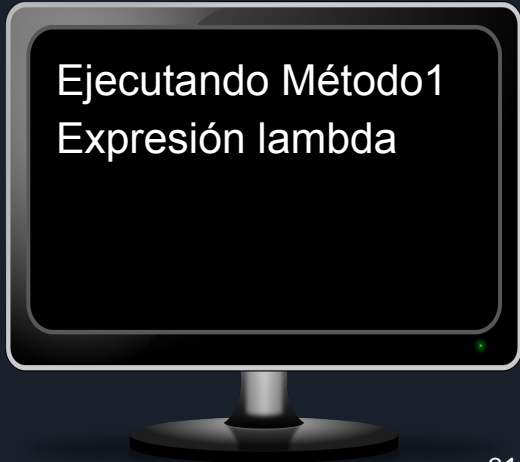


```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action? a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        a -= Metodo2;
        a?.Invoke();
    }
    . . .
}
```

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

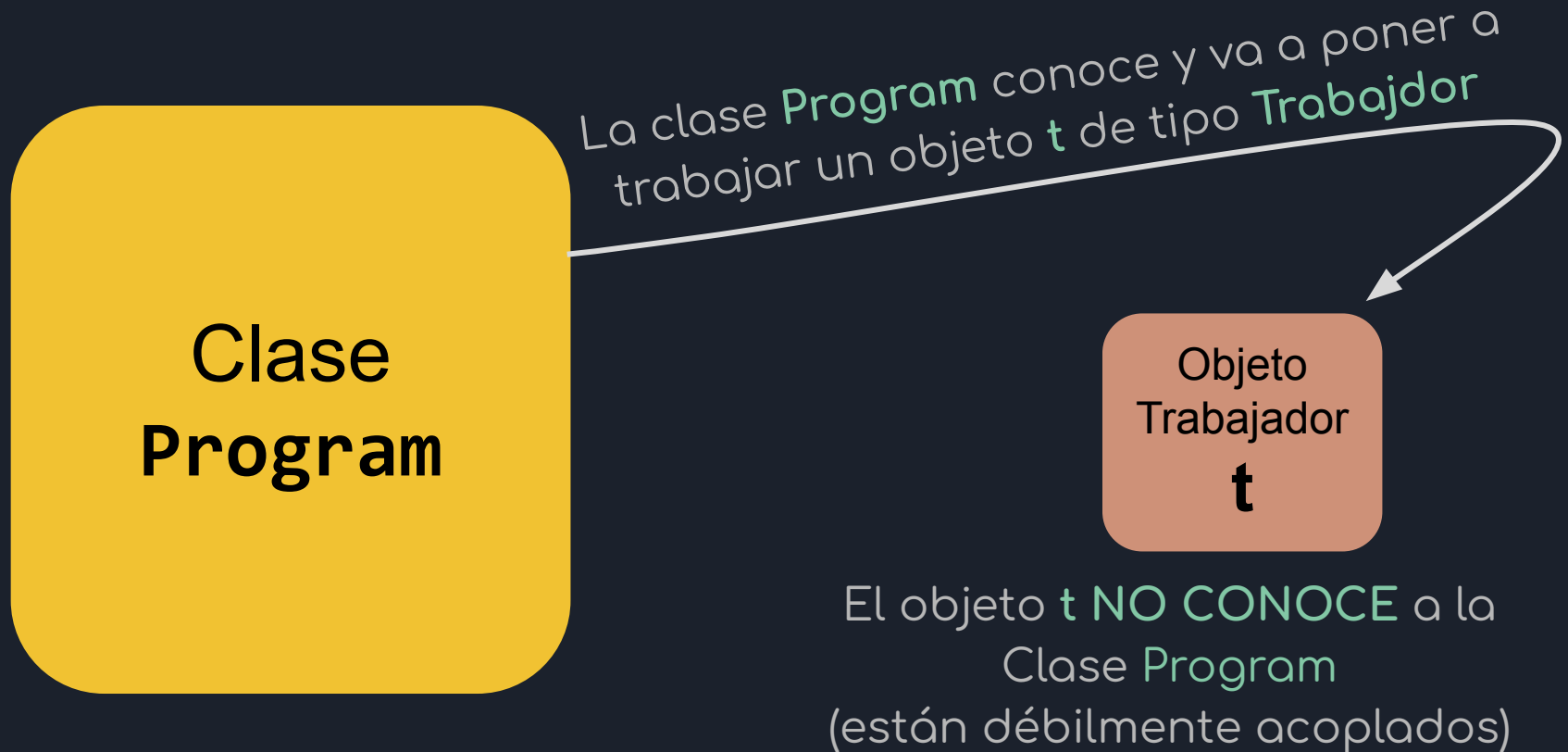
Eventos

- 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)
```

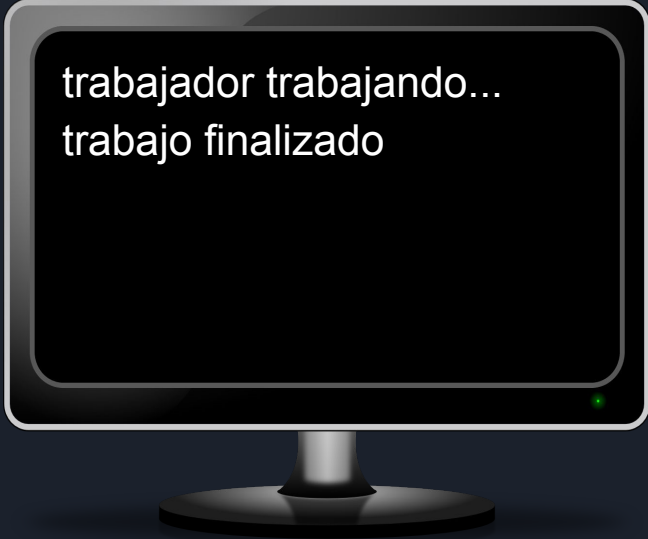
```
        {
```

```
            TrabajoFinalizado();
```

```
        }
```

```
    }
```

```
}
```



trabajador trabajando...
trabajo finalizado

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-----

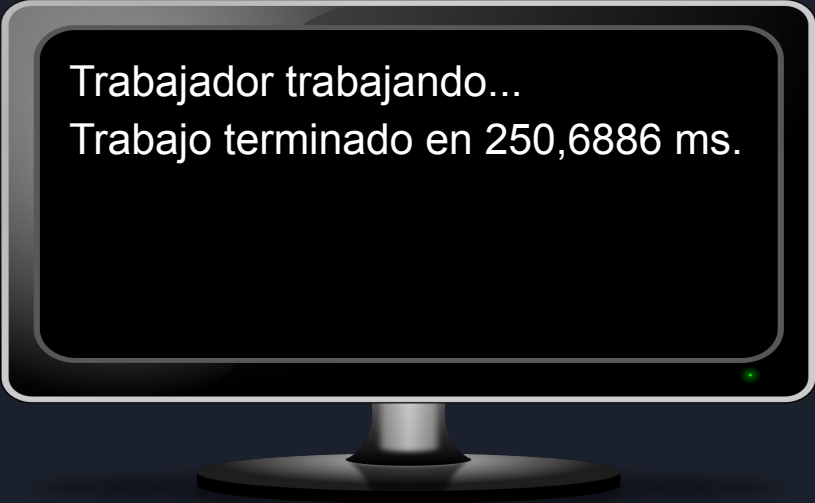
```
class Trabajador
{
    public TrabajoFinalizadoEventHandler? TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("Trabajador trabajando...");
        DateTime tInicial = DateTime.Now;
        for (int i = 1; i < 100_000_000; i++) ; //Pierdo tiempo simulando el trabajo
        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

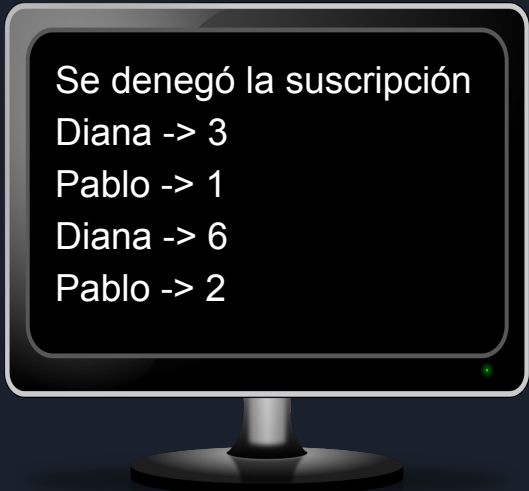
-----Program.cs-----

```
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 del material
complementario