



.NET

Teoría 8

Más sobre Interfaces Herencia

Interfaces - herencia

Las interfaces pueden heredar de múltiples interfaces


```
interface IRegistrable {  
    void Registrar();  
}
```

```
interface IReportable {  
    void GenerarReporte();  
}
```

```
interface IAuditable: IRegistrable, IReportable {  
    DateTime ObtenerUltimaAuditoria();  
}
```

```
class AccionAuditable : IAuditable {  
    . . .  
}
```

La clase *AccionAuditable* debe implementar
Registrar() (de IRegistrable),
GenerarReporte() (de IReportable)
y ObtenerUltimaAuditoria() (de IAuditable)



Implementando múltiples Interfaces

```
interface IImprimible
{
    void Imprimir();
}
```

```
interface IGuardable
{
    void Guardar(string rutaArchivo);
}
```

```
class Documento : IImprimible, IGuardable
{
    . . .
}
```

La clase Documento debe implementar `Imprimir()` (de `IImprimible`) y `Guardar()` (de `IGuardable`).

Implementando Interfaces con miembros duplicados

```
interface ITareaEjecutable {  
    void Iniciar();  
    void Detener();  
}
```

```
interface ITemporizador {  
    void Iniciar();  
    TimeSpan ObtenerTiempoTranscurrido();  
}
```

```
class TareaConTemporizador : ITareaEjecutable, ITemporizador  
{  
    public void Iniciar()  
    {  
        . . .  
    }  
    . . .  
}
```

Una única implementación de **Iniciar()**
cumple ambos contratos.
Sin embargo, ¿Es claro qué debe "Iniciar"
este método?

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 TareaConTemporizador : ITareaEjecutable, ITemporizador
{
```



```
void ITareaEjecutable.Iniciar()
```

```
{
```

```
    . . . // este método debería iniciar la tarea
```

```
}
```



```
void ITemporizador.Iniciar()
```

```
{
```

```
    . . . // este método debería iniciar el temporizador
```

```
}
```

```
public void Iniciar()
```

```
{
```

```
    . . . // podría usar este método para iniciar la tarea y el temporizador
```

```
}
```

```
    . . .
```

```
}
```

Atención

Los métodos en las implementaciones explícitas de interfaces no llevan el modificador de acceso `public`

Implementación explícita de miembros de interfaces

Invocando las distintas implementaciones del método Iniciar

```
. . .  
TareaConTemporizador tarea = new TareaConTemporizador();  
. . .  
(tarea as ITemporizador).Iniciar(); //invoca la implementación explícita  
                                     //de ITemporizador  
. . .  
(tarea as ITareaEjecutable).Iniciar(); //invoca la implementación explícita  
                                         //de ITareaEjecutable  
. . .  
tarea.Iniciar(); //invoca la implementación a nivel de 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

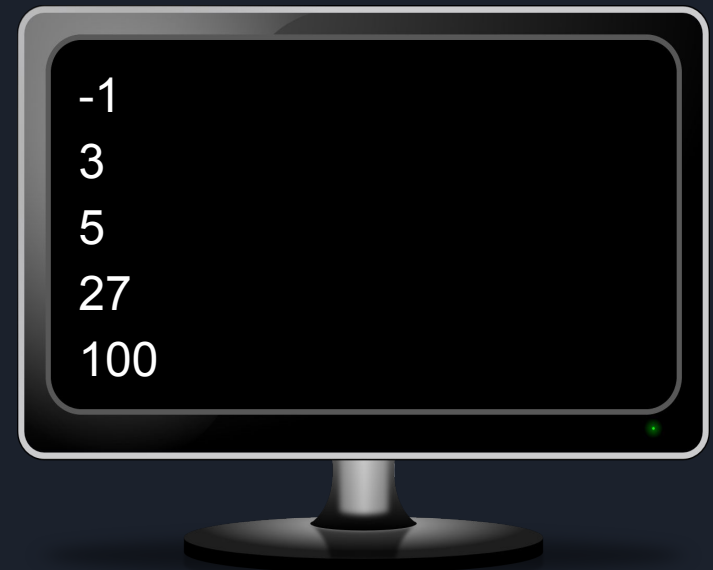
1. una implementación a nivel de clase
2. una implementación explícita de interface
3. Ambas, una implementación explícita de interface y una implementación a nivel de clase

Interfaces de la plataforma que se usan para la comparación

Interface IComparable. Ordenamiento - Ejemplo 1

```
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. Ordenamiento - Ejemplo 1

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(string nombre, int legajo)
{
    public string Nombre { get; } = nombre;
    public int Legajo { get; } = legajo;
    public void Imprimir()
        => Console.WriteLine($"Soy {Nombre}, legajo {Legajo}");
}
```



Ordenamiento - Ejemplo 2

Codificar Program.cs de la siguiente manera



```
using Teoria8;

Empleado[] empleados = [
    new Empleado("Juan",79),
    new Empleado("Adriana",123),
    new Empleado("Diego",23)
];
Array.Sort(empleados);
foreach (Empleado e in empleados)
{
    e.Imprimir();
}
```


Interfaces - System.IComparable

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`

```
Program.cs
1
2
3 Empleado[] empleados
4     new Empleado("J
5     new Empleado("Adriana",123),
6     new Empleado("Diego",23)
7 ];
8 Array.Sort(empleados);
```

Exception has occurred: CLR/System.InvalidOperationException ×

An unhandled exception of type 'System.InvalidOperationException' occurred in System.Private.CoreLib.dll: 'Failed to compare two elements in the array.'

Inner exceptions found, see \$exception in variables window for more details.

Innermost exception System.ArgumentException : At least one object must implement IComparable.

```
at System.Collections.Comparer.Compare(Object a, Object b)
at System.Collections.Generic.ArraySortHelper`1.SwapIfGreater(Span`1 keys, Comparison`1 comparer, Int32 i, Int32 j)
at System.Collections.Generic.ArraySortHelper`1.IntroSort(Span`1 keys, Int32 depthLimit, Comparison`1 comparer)
at System.Collections.Generic.ArraySortHelper`1.IntrospectiveSort(Span`1 keys, Comparison`1 comparer)
at System.Collections.Generic.ArraySortHelper`1.Sort(Span`1 keys, IComparer`1 comparer)
```

```
9 foreach (Empleado e in empleados)
10 {
11     e.Imprimir();
12 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter (e.g. text, !exclude)

Loaded '/home/leo/dotnet/shared/Microsoft.NETCore.App/8.0.0/System.Text.Encoding.Extensions.dll'. Skipped loading symbols. Module is optimized and the debugger option 'Just My Code' is enabled.

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



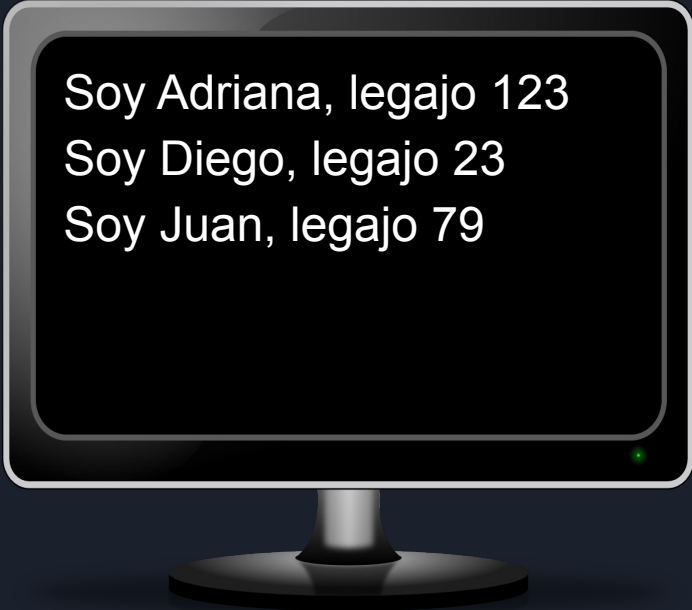
```
class Empleado(string nombre, int legajo) : IComparable
{
    public int CompareTo(object? obj)
    {
        int result = 0;
        if (obj is Empleado e)
        {
            string nombre = e.Nombre;
            result = this.Nombre.CompareTo(nombre);
        }
        return result;
    }
}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

Ordenamiento Ejemplo 2



```
using Teoria8;

Empleado[] empleados = [
    new Empleado("Juan",79),
    new Empleado("Adriana",123),
    new Empleado("Diego",23)
];
Array.Sort(empleados);
foreach (Empleado e in empleados)
{
    e.Imprimir();
}
```



Soy Adriana, legajo 123
Soy Diego, legajo 23
Soy Juan, legajo 79

Interface IComparer

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

```
namespace Teoria8;

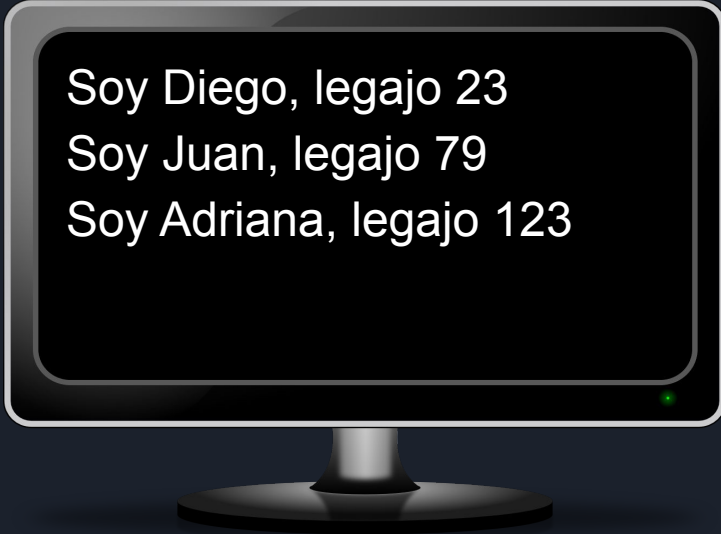
class ComparadorPorLegajo : System.Collections.IComparer
{
    public int Compare(object? x, object? y)
    {
        int result = 1;
        if (x is Empleado e1 && y is Empleado e2)
        {
            int legajo1 = e1.Legajo;
            int legajo2 = e2.Legajo;
            result = legajo1.CompareTo(legajo2);
        }
        return result;
    }
}
```

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


```
using Teoria8;
```

```
Empleado[] empleados = [  
    new Empleado("Juan",79),  
    new Empleado("Adriana",123),  
    new Empleado("Diego",23)  
];  
Array.Sort(empleados, new ComparadorPorLegajo());  
foreach (Empleado e in empleados)  
{  
    e.Imprimir();  
}
```

Ordenamiento
por legajo



Soy Diego, legajo 23
Soy Juan, legajo 79
Soy Adriana, legajo 123

Ordenamiento - Ejemplo 4

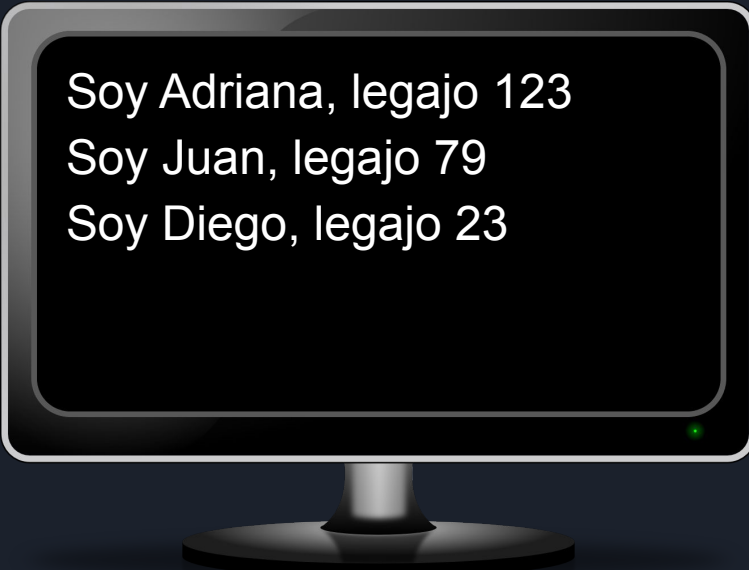
```
class ComparadorPorLegajo : System.Collections.IComparer
{
    public bool Descendente { get; set; } = false;
    public int Compare(object? x, object? y)
    {
        int result = 1;
        if (x is Empleado e1 && y is Empleado e2)
        {
            int legajo1 = e1.Legajo;
            int legajo2 = e2.Legajo;
            result = legajo1.CompareTo(legajo2);
        }
        if (Descendente)
        {
            result = -result;
        }
        return result;
    }
}
```

Modificando
ComparadorPorLegajo para
permitir ordenar ascendente o
descendentemente

```
using Teoria8;

Empleado[] empleados = [
    new Empleado("Juan",79),
    new Empleado("Adriana",123),
    new Empleado("Diego",23)
];

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



Soy Adriana, legajo 123
Soy Juan, legajo 79
Soy Diego, legajo 23

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 =["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 unaPyme = new Pyme(new Empleado("Juan",79),  
                          new Empleado("Adriana",123),  
                          new Empleado("Diego",23));
```

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

Error de compilación

```
using Teoria8;
```

```
Pyme unaPyme = new Pyme(new Empleado("Juan",79),  
                        new Empleado("Adriana",123),  
                        new Empleado("Diego",23));
```

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

Error de compilación:
'Pyme' no contiene ninguna definición de
extensión o instancia pública para
'GetEnumerator'
unaPyme 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;
    }
    public IEnumerator GetEnumerator()
    {
        return empleados.GetEnumerator();
    }
}
```

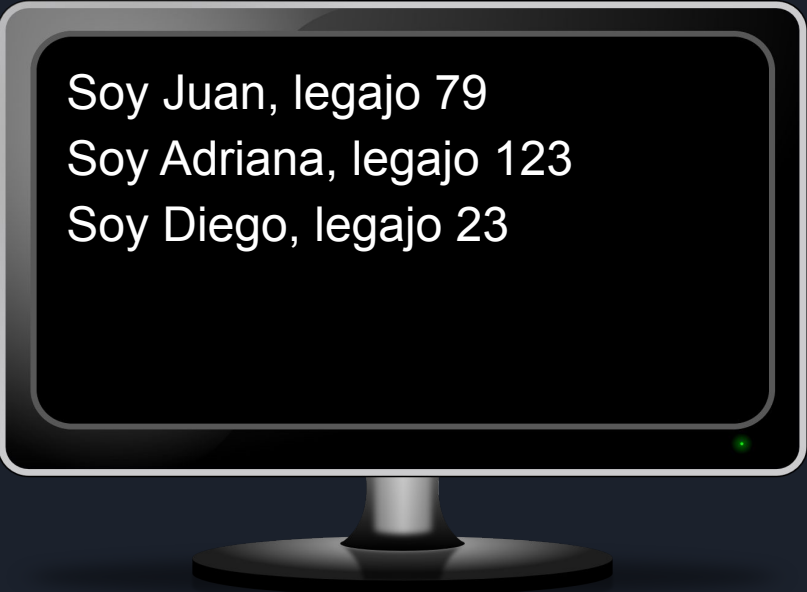
Los arreglos implementan la interface `IEnumerable`, estamos aprovechando el enumerador que proveen

```
using Teoria8;

Pyme unaPyme = new Pyme(new Empleado("Juan", 79),
                        new Empleado("Adriana", 123),
                        new Empleado("Diego", 23));

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

Solucionado !



Soy Juan, legajo 79
Soy Adriana, legajo 123
Soy Diego, legajo 23

¿ 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();

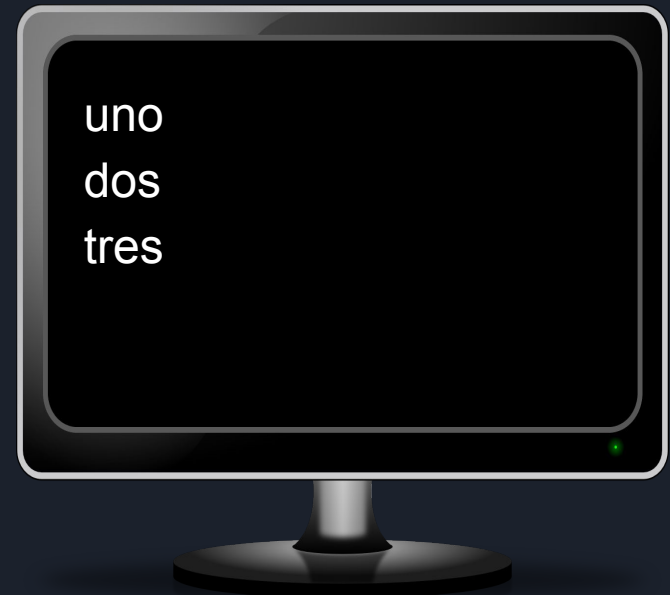
        // Sets the enumerator before the first element
        void Reset();
    }
}
```

Recorriendo un enumerador

```
using System.Collections;

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

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



Recorriendo un enumerador

```
using System.Collections;
```

```
string[] vector = ["uno", "dos", "tres"];
```

```
IEnumerator e = vector.GetEnumerator();
```

```
while (e.MoveNext())
```

```
{
```

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

```
}
```



inválido

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

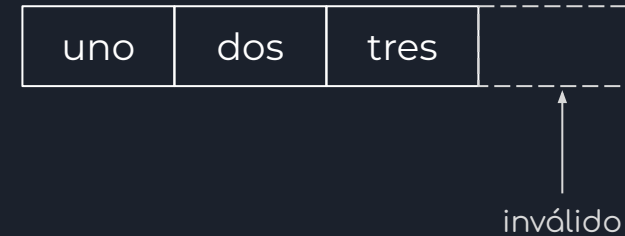
Tip: Sólo invocar `e.Current` luego de obtener `true` con `e.MoveNext()`

Recorriendo un enumerador

```
using System.Collections;

string[] vector = ["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);
}
```



Verano
Otoño
Invierno
Primavera

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 uso de `yield return`

```
using System.Collections;
```

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

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

```
{
```

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

```
}
```

Current es de tipo
object

```
IEnumerator colores()
```

```
{
```

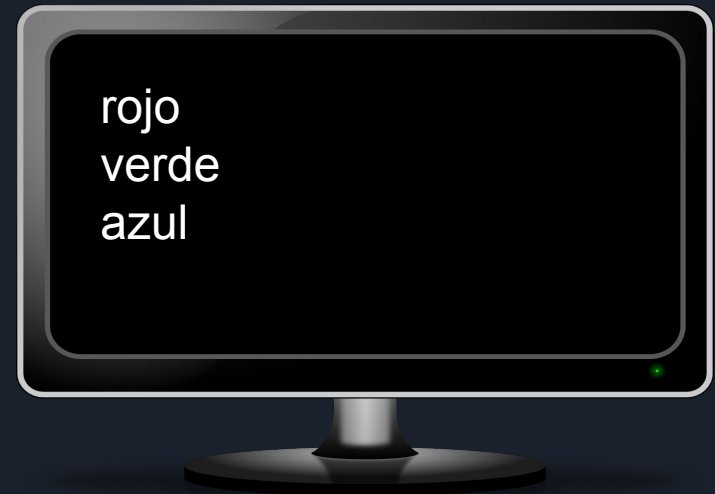
```
    yield return "rojo";
```

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

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

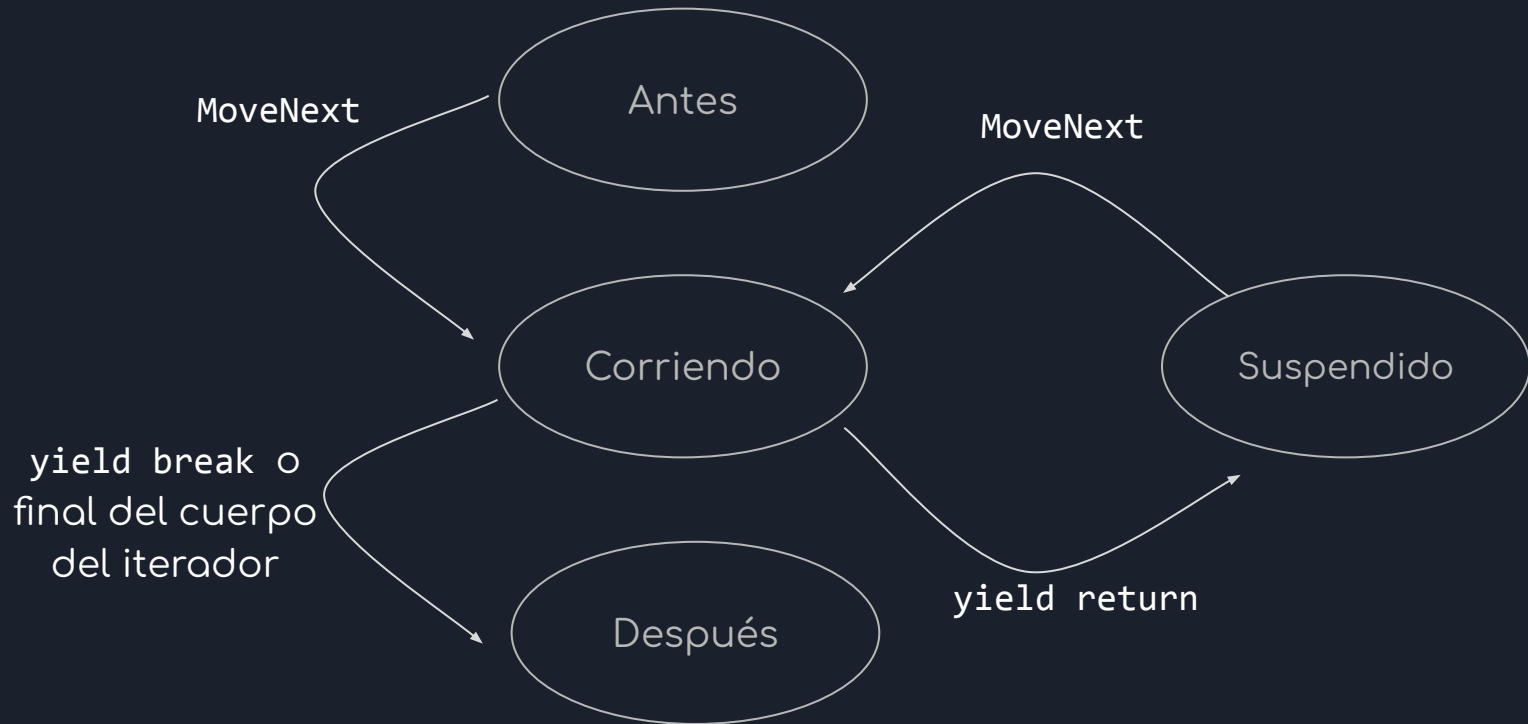
```
}
```

Este método es
un iterador



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



Esto es parte del código que genera el compilador a partir del código de la diapositiva 50

Clase para el iterador



```
internal class Program
```

```
{
```

```
    private sealed class <<<Main>>>g__colores|0_0>d  
    {
```

```
        private int <>1__state;  
        private object <>2__current;  
        object IEnumerator.Current  
        {
```

```
            get  
            {  
                return <>2__current;  
            }  
        }
```

```
        object IEnumerator.Current  
        {
```

```
            get  
            {  
                return <>2__current;  
            }  
        }
```

```
        public <<<Main>>>g__colores|0_0>d(int <>1__state)  
        {  
            this.<>1__state = <>1__state;  
        }
```

```
        void IDisposable.Dispose()  
        {  
        }
```

```
        bool IEnumerator.MoveNext()  
        {  
            return this.MoveNext();  
        }
```

```
        void IEnumerator.Reset()  
        {  
            throw new NotSupportedException();  
        }
```

Constructor



```
private bool MoveNext()  
{  
    switch (<>1__state)  
    {  
        default:  
            return false;  
        case 0:  
            <>1__state = -1;  
            <>2__current = "rojo";  
            <>1__state = 1;  
            return true;  
        case 1:  
            <>1__state = -1;  
            <>2__current = "verde";  
            <>1__state = 2;  
            return true;  
        case 2:  
            <>1__state = -1;  
            <>2__current = "azul";  
            <>1__state = 3;  
            return true;  
        case 3:  
            <>1__state = -1;  
            return false;  
    }  
}
```

```
private static void <Main>$(string[] args)  
{  
    IEnumerator enumerator = <<Main>>g__colores|0_0();  
    while (enumerator.MoveNext())  
    {  
        Console.WriteLine(enumerator.Current);  
    }  
}  
internal static IEnumerator <<Main>>g__colores|0_0()  
{  
    return new <<<Main>>>g__colores|0_0>d(0);  
}
```

Esto es parte del código que genera el compilador a partir del código de la diapositiva 50

Clase para el iterador



```
internal class Program
```

```
{  
    private sealed class <<<Main>>>g__colores|0_0>d
```

```
{  
    private int <>1__state;  
    private object <>2__current;  
    object IEnumerator.Current
```

```
{  
        get  
        {  
            return <>2__current;  
        }  
    }  
    object IEnumerator.Current
```

```
{  
        get  
        {  
            return <>2__current;  
        }  
    }  
}
```

Constructor



```
public <<<Main>>>g__colores|0_0>d(int <>1__state)  
{  
    this.<>1__state = <>1__state;  
}
```

```
void IDisposable.Dispose()  
{  
}
```

```
bool IEnumerator.MoveNext()
```

```
{  
    return this.MoveNext();  
}
```

```
void IEnumerator.Reset()
```

```
{  
    throw new NotSupportedException();  
}
```

!!! Cuidado !!!



```
private bool MoveNext()  
{  
    switch (<>1__state)  
    {  
        default:  
            return false;  
        case 0:  
            <>1__state = -1;  
            <>2__current = "rojo";  
            <>1__state = 1;  
            return true;  
        case 1:  
            <>1__state = -1;  
            <>2__current = "verde";  
            <>1__state = 2;  
            return true;  
        case 2:  
            <>1__state = -1;  
            <>2__current = "azul";  
            <>1__state = 3;  
            return true;  
        case 3:  
            <>1__state = -1;  
            return false;  
    }  
}
```

```
private static void <Main>$(string[] args)
```

```
{  
    IEnumerator enumerator = <<Main>>g__colores|0_0();  
    while (enumerator.MoveNext())
```

```
{  
        Console.WriteLine(enumerator.Current);  
    }  
}
```

```
internal static IEnumerator <<Main>>g__colores|0_0()
```

```
{  
    return new <<<Main>>>g__colores|0_0>d();  
}
```

```
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()**

Interfaces - Iteradores

```
using System.Collections;
```

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

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

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

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

```
IEnumerator colores()  
{  
    yield return "rojo";  
    yield return "verde";  
    yield return "azul";  
}
```

Curiosidad

Un enumerador generado con un iterador no lanza una excepción `InvalidOperationException` en estos puntos críticos

devuelve null



null
rojo
verde
azul
azul

Sigue devolviendo el último elemento

```
using System.Collections;
```

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

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

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

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

```
IEnumerator colores()  
{  
    yield return "rojo";  
    yield return "verde";  
    yield return "azul";  
}
```

No importa cómo se generó,
esta es siempre la forma
correcta de recorrer un
enumerador

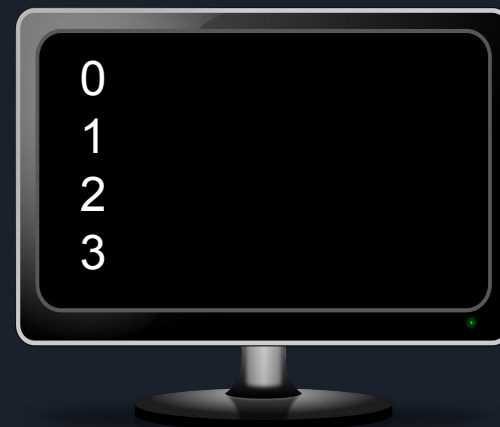


Iteradores - ejemplo 2 uso de `yield break`

```
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

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 Letras_A())
{
    Console.WriteLine(st);
    if (st == "AAAA")
    {
        break;
    }
}
```

```
IEnumerable Letras_A()
{
    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 Teoria-Recursos.txt

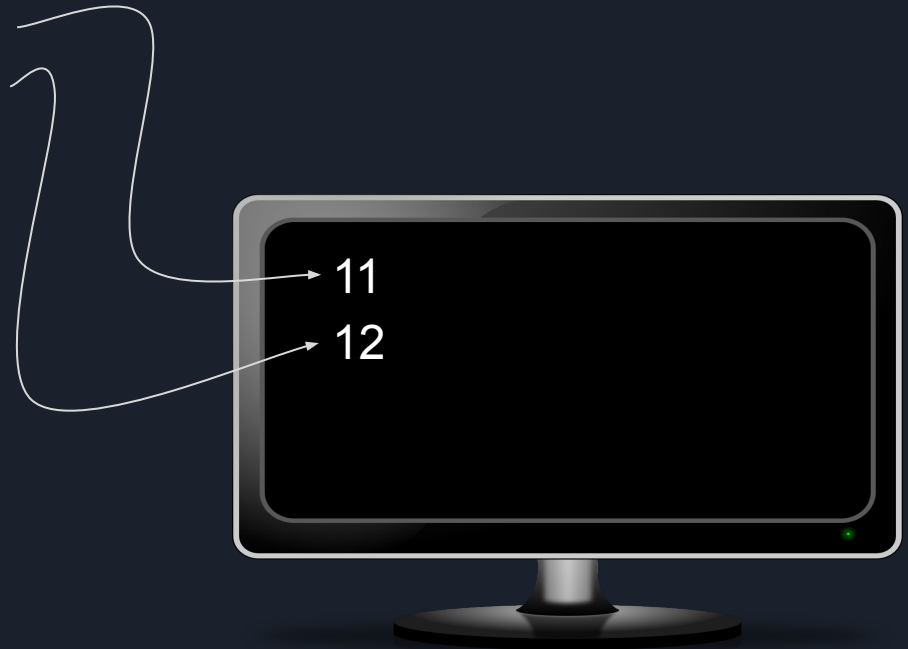
----- Auxiliar.cs -----

```
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;  
}
```

Código en el archivo
08 Teoria-Recursos.txt

Delegados - Introducción

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

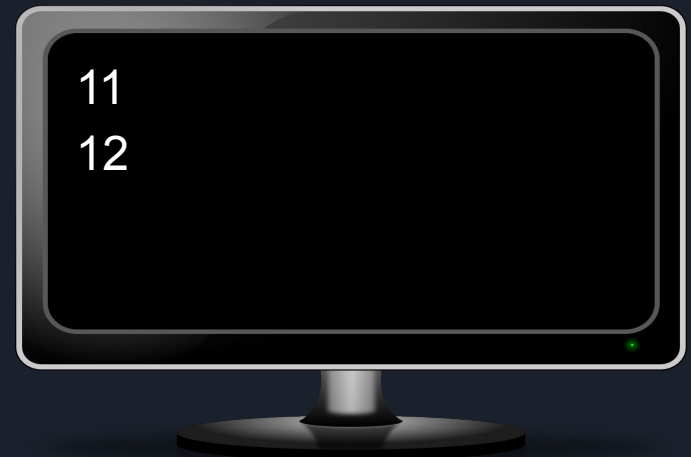
```
namespace Teoria8;  
delegate int FuncionEntera(int n);
```

-----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



. . .

```
List<int> Seleccionar(List<int> lista, FuncionEntera f)
{
    var result = new List<int>();
    foreach (int valor in lista)
    {
        result.Add(f(valor));
    }
    return result;
}
```

Recibe como parámetros una lista y una función en un delegado

```
void Imprimir(List<int> lista)
{
    foreach (int i in lista)
    {
        Console.Write(i + " ");
    }
    Console.WriteLine();
}
```

Construye una nueva lista aplicando la función f a cada uno de los elementos de la lista recibida



Modificar el método Procesar de la clase Auxiliar y ejecutar



```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        List<int> lista = [11, 5, 90];
        var lista1 = Seleccionar(lista, SumaUno);
        Imprimir(lista1);
        var lista2 = Seleccionar(lista, SumaDos);
        Imprimir(lista2);
    }

    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;
    List<int> Seleccionar(List<int> lista, FuncionEntera f)
    {
        . . .
    }
}
```

Delegados - Pasar métodos como parámetros

```
namespace Teoria8;

public class Auxiliar
{
    public void Procesar()
    {
        List<int> lista = [11, 5, 90];
        var lista1 = Seleccionar(lista, SumaUno);
        Imprimir(lista1);
        var lista2 = Seleccionar(lista, SumaDos);
        Imprimir(lista2);
    }

    int SumaUno(int n) => n + 1;

    int SumaDos(int n) => n + 2;

    List<int> Seleccionar(List<int> lista, FuncionEntera f)
    {
        var result = new List<int>();
        foreach (int valor in lista)
        {
            result.Add(f(valor));
        }
        return result;
    }

    void Imprimir(List<int> lista)
    {
        foreach (int i in lista)
        {
            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()
    {
        List<int> lista = [11, 5, 90];
        FuncionEntera f = delegate (int n)
        {
            return n * 2;
        };
        var lista1 = Seleccionar(lista, f);
        Imprimir(lista1);
        var lista2 = Seleccionar(lista, delegate (int n) { return n + 10; });
        Imprimir(lista2);
    }
}
```

...

Delegados - Métodos anónimos

```
namespace Teoria8;

public class Auxiliar
{
    public void Procesar()
    {
        List<int> lista = [11, 5, 90];
        FuncionEntera f = delegate (int n)
        {
            return n * 2;
        };
        var lista1 = Seleccionar(lista, f);
        Imprimir(lista1);
        var lista2 = Seleccionar(lista, delegate (int n) { return n + 10; });
        Imprimir(lista2);
    }

    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;

    List<int> Seleccionar(List<int> lista, FuncionEntera f)
    {
        var result = new List<int>();
        foreach (int valor in lista)
        {
            result.Add(f(valor));
        }
        return result;
    }

    void Imprimir(List<int> lista)
    {
        foreach (int i in lista)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();
    }
}
```



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 (incluso si fuese de retorno), puede reemplazarse todo el bloque por esa expresión:

```
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()
    {
        List<int> lista = [11, 5, 90];
        var lista1 = Seleccionar(lista, n => n * 2);
        Imprimir(lista1);
        var lista2 = Seleccionar(lista, n => n + 10);
        Imprimir(lista2);
    }
}
```

. . .

Delegados - Expresiones lambda

```
namespace Teoria8;

public class Auxiliar
{
    public void Procesar()
    {
        List<int> lista = [11, 5, 90];
        var lista1 = Seleccionar(lista, n => n * 2);
        Imprimir(lista1);
        var lista2 = Seleccionar(lista, n => n + 10);
        Imprimir(lista2);
    }

    int SumaUno(int n) => n + 1;
    int SumaDos(int n) => n + 2;

    List<int> Seleccionar(List<int> lista, FuncionEntera f)
    {
        var result = new List<int>();
        foreach (int valor in lista)
        {
            result.Add(f(valor));
        }
        return result;
    }

    void Imprimir(List<int> lista)
    {
        foreach (int i in lista)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();
    }
}
```



Métodos que devuelven delegados

- Los delegados también pueden utilizarse como tipo de propiedades y como tipo de retorno de los métodos.
- Si tenemos este delegado:

```
delegate int Operacion(int a, int b);
```

- Podríamos definir este método

```
// Método que devuelve un delegado del tipo Operacion
Operacion ObtenerOperacion(string tipoOperacion)
{
    switch (tipoOperacion)
    {
        case "sumar":
            return (a, b) => a + b; // Devuelve una expresión lambda que suma
        case "restar":
            return (a, b) => a - b; // Devuelve una expresión lambda que resta
        default:
            return (a, b) => 0;
    }
}
```

Métodos que devuelven delegados

- Luego, podríamos invocarlo de la siguiente manera:

```
// Obtiene un delegado para la operación de suma
Operacion suma = ObtenerOperacion("sumar");

// Llama al delegado para realizar la suma
int resultadoSuma = suma(10, 5);
Console.WriteLine($"Suma: {resultadoSuma}"); // Imprime "Suma: 15"

// Obtiene un delegado para la operación de resta
Operacion resta = ObtenerOperacion("restar");

// Llama al delegado para realizar la resta
int resultadoResta = resta(10, 5);
Console.WriteLine($"Resta: {resultadoResta}"); // Imprime "Resta: 5"
```


Fin de la teoría 8



- **Preferir `yield` para Enumerables Personalizados:**

- Para crear una secuencia personalizada (que funcione con `foreach`), **casi siempre** es más simple, legible y menos propenso a errores usar `yield return` dentro de un método que devuelva `IEnumerable`. Evitar implementar `IEnumerable/IEnumerator` manualmente a menos que sea estrictamente necesario (casos muy avanzados).

- **Evitar `Reset()` en Enumeradores:**

- No confiar en el método `Reset()` de `IEnumerator`. Muchos enumeradores (incluidos los generados por `yield`) no lo implementan o lanzan `NotSupportedException`. Si se necesita reiniciar una nueva iteración, obtener un nuevo enumerador.

- **Usar Delegados para Pasar Comportamiento:**

- Pensar en delegados cuando se necesite pasar un método como parámetro a otro método (ej: para configurar una devolución de llamada (callback), una estrategia de procesamiento, un criterio de filtrado/ordenamiento).

- **Preferir Expresiones Lambda por simplicidad y legibilidad:**

- Para asignar código a un delegado, las **expresiones lambda (`=>`)** son la forma preferida en C# moderno. Son más concisas y legibles que los métodos anónimos.



Práctica sobre la teoría 8

Práctica sobre la teoría 8

1) Tomar como base el ejercicio 2 de la práctica 7 e incorporar las interfaces, propiedades y métodos necesarios para que el siguiente código produzca la salida indicada:

```
INombrable[] vector = [  
    new Persona() {Nombre="Zulema"},  
    new Perro() {Nombre="Sultán"},  
    new Persona() {Nombre="Claudia"},  
    new Persona() {Nombre="Carlos"},  
    new Perro() {Nombre="Chopper"},  
];  
Array.Sort(vector); //debe ordenar por Nombre alfabéticamente  
foreach (INombrable n in vector)  
{  
    Console.WriteLine($"{n.Nombre}: {n}");  
}
```

Salida por consola

```
Carlos: Carlos es una persona  
Chopper: Chopper es un perro  
Claudia: Claudia es una persona  
Sultán: Sultán es un perro  
Zulema: Zulema es una persona
```

2) Modificar el ejercicio anterior para que el siguiente código produzca la salida indicada:

```
INombrable[] vector = [  
    new Persona() {Nombre="Ana María"},  
    new Perro() {Nombre="Sultán"},  
    new Persona() {Nombre="Ana"},  
    new Persona() {Nombre="José Carlos"},  
    new Perro() {Nombre="Chopper"}  
];  
Array.Sort(vector, new ComparadorLongitudNombre()); //ordena por longitud de Nombre  
foreach (INombrable n in vector)  
{  
    Console.WriteLine($"{n.Nombre.Length}: {n.Nombre}");  
}
```

Salida por consola

```
3: Ana  
6: Sultán  
7: Chopper  
9: Ana María  
11: José Carlos
```

3) Codificar usando iteradores los métodos:

Rango(i, j, p) que devuelve la secuencia de enteros desde **i** hasta **j** con un paso de **p**

Potencia(b,k) que devuelve la secuencia b^1, b^2, \dots, b^k

DivisiblePor(e,i) retorna los elementos de **e** que son divisibles por **i**

Observar la salida que debe producir el siguiente código:

```
using System.Collections;

IEnumerable rango = Rango(6, 30, 3);
IEnumerable potencias = Potencias(2, 10);
IEnumerable divisibles = DivisiblesPor(rango, 6);
foreach (int i in rango)
{
    Console.Write(i + " ");
}
Console.WriteLine();
foreach (int i in potencias)
{
    Console.Write(i + " ");
}
Console.WriteLine();
foreach (int i in divisibles)
{
    Console.Write(i + " ");
}
Console.WriteLine();
```

Salida por consola

```
6 9 12 15 18 21 24 27 30
2 4 8 16 32 64 128 256 512 1024
6 12 18 24 30
```

4) Declarar los tipos delegados necesarios para que el siguiente programa compile y produzca la salida en la consola indicada

```
Del1 d1 = delegate (int x) { Console.WriteLine(x); };  
d1(10);
```

```
Del2 d2 = x => Console.WriteLine(x.Length);  
d2(new int[] { 2, 4, 6, 8 });
```

```
Del3 d3 = x =>  
{  
    int sum = 0;  
    for (int i = 1; i <= x; i++)  
    {  
        sum += i;  
    }  
    return sum;  
};
```

```
int resultado = d3(10);  
Console.WriteLine(resultado);
```

```
Del4 d4 = new Del4(LongitudPar);  
Console.WriteLine(d4("hola mundo"));
```

```
bool LongitudPar(string st)  
{  
    return st.Length % 2 == 0;  
}
```

Salida por consola

```
10  
4  
55  
True
```

Práctica sobre la teoría 8

5) ¿Qué obtiene un método anónimo (o expresión lambda) cuando accede a una variable definida en el entorno que lo rodea, una copia del valor de la variable o la referencia a dicha variable? Tip: Observar la salida por consola del siguiente código:

```
int i = 10;
Action a = delegate ()
{
    Console.WriteLine(i);
};
a.Invoke();
i = 20;
a.Invoke();
```

6) Teniendo en cuenta lo respondido en el ejercicio anterior, ¿Qué salida produce en la consola la ejecución del siguiente programa?

```
Action[] acciones = new Action[10];
for (int i = 0; i < 10; i++)
{
    acciones[i] = () => Console.Write(i + " ");
}
foreach (var a in acciones)
{
    a.Invoke();
}
```

Práctica sobre la teoría 8

7) En este ejercicio, se requiere extender el tipo `int[]` con algunos métodos de extensión. Se presenta el código del método de extensión `Print(this int[] vector, string leyenda)` que imprime en la consola los elementos del vector precedidos por una leyenda que se pasa como parámetro. Se requiere codificar el método de extensión `Seleccionar(...)` que recibe como parámetro un delegado de tipo `FuncionEntera` y devuelve un nuevo vector de enteros producto de aplicar la función recibida como parámetro a cada uno de los elementos del vector. El siguiente programa debe producir la salida indicada.

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

```
int[] vector = [1, 2, 3, 4, 5];  
vector.Print("Valores iniciales: ");  
var vector2 = vector.Seleccionar(n => n * 3);  
vector2.Print("Valores triplicados: ");  
vector.Seleccionar(n => n * n).Print("Cuadrados: ");
```

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

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

Salida por consola

```
Valores iniciales: 1, 2, 3, 4, 5  
Valores triplicados: 3, 6, 9, 12, 15  
Cuadrados: 1, 4, 9, 16, 25
```

Para ello, completar el código de la siguiente clase estática `VectorDeEnterosExtension`

```
static class VectorDeEnterosExtension  
{  
    public static void Print(this int[] vector, string leyenda)  
    {  
        Console.WriteLine(leyenda + string.Join(", ", vector));  
    }  
    public static int[] Seleccionar(. . . ) {  
        . . .  
    }  
}
```


8) Agregar al ejercicio anterior el método de extensión **Donde(...)** para el tipo **int[]** que recibe como parámetro un delegado de tipo **Predicado** y devuelve un nuevo vector de enteros con los elementos del vector que cumplen ese predicado. El siguiente programa debe producir la salida indicada.

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

```
int[] vector =[1, 2, 3, 4, 5];  
vector.Print("Valores iniciales: ");  
vector.Donde(n => n % 2 == 0).Print("Pares: ");  
vector.Donde(n => n % 2 == 1).Seleccionar(n => n * n).Print("Impares al cuadrado: ");
```

```
-----Predicado.cs-----
```

```
delegate bool Predicado(int n);
```

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

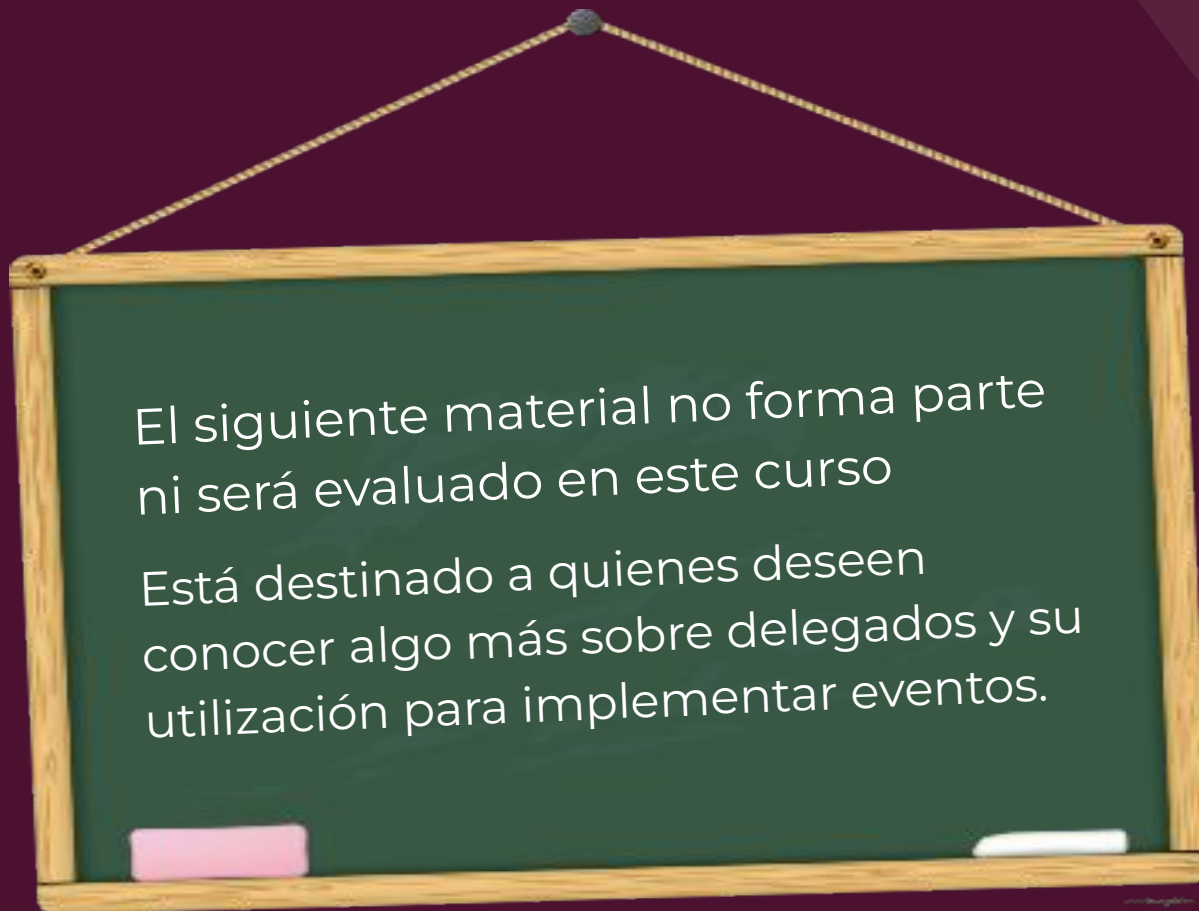
```
delegate int FuncionEntera(int n);
```

Salida por consola

```
Valores iniciales: 1, 2, 3, 4, 5  
Pares: 2, 4  
Impares al cuadrado: 1, 9, 25
```

Para conocer más

Contenido optativo





Modificar la clase Auxiliar de la teoría 8 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");
    . . .
}
```

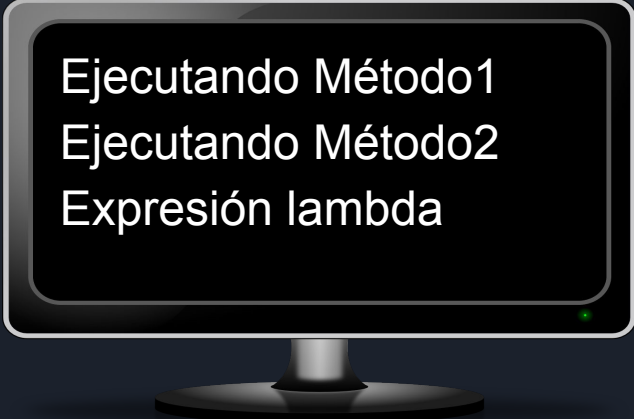
Action es un delegado ya definido en la BCL de la siguiente manera:
`public delegate void Action();`

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

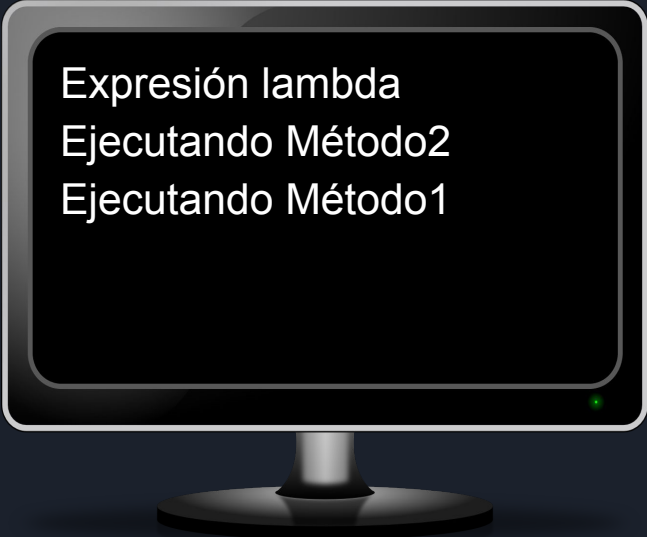
```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        Delegate[] encolados = a.GetInvocationList();
        for (int i = encolados.Length - 1; i >= 0; i--)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

    static void Metodo1()
        => Console.WriteLine("Ejecutando Método1");
    static void Metodo2()
        => Console.WriteLine("Ejecutando Método2");
}
```

...

Invocando a los
delegados en
orden inverso

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



Expresión lambda
Ejecutando Método2
Ejecutando Método1



Modificar el método Procesar() de la clase Auxiliar

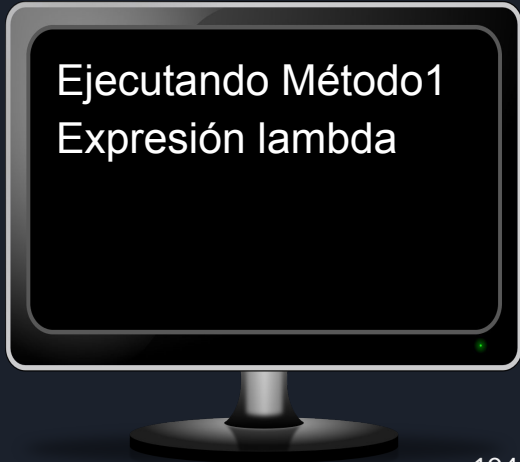


```
namespace Teoria8;
class Auxiliar
{
    public void Procesar()
    {
        Action? a;
        a = Metodo1;
        a = a + Metodo2;
        a += () => Console.WriteLine("Expresión lambda");
        a -= Metodo2;
        a?.Invoke();
    }
    . . .
}
```

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

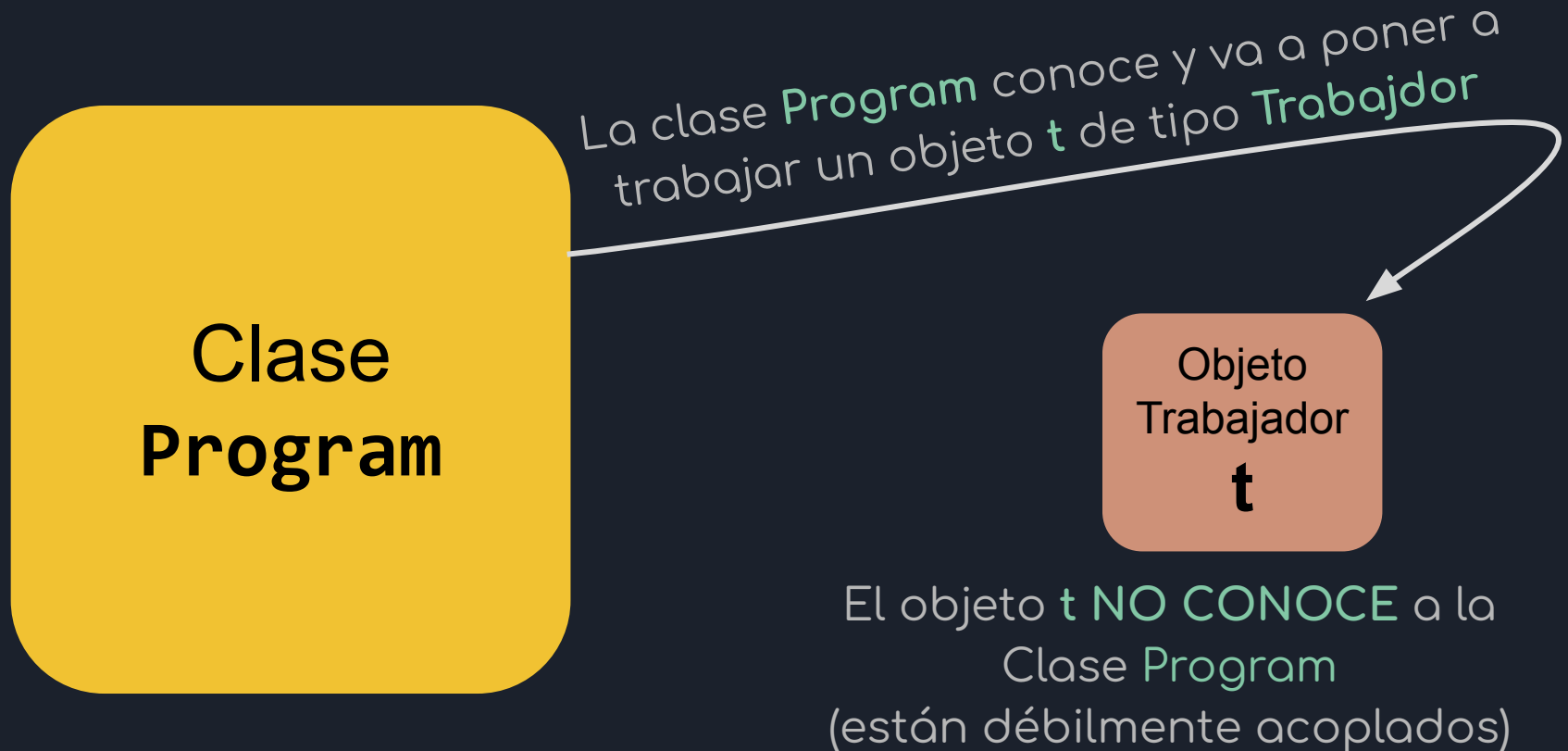
Eventos

- 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)
```

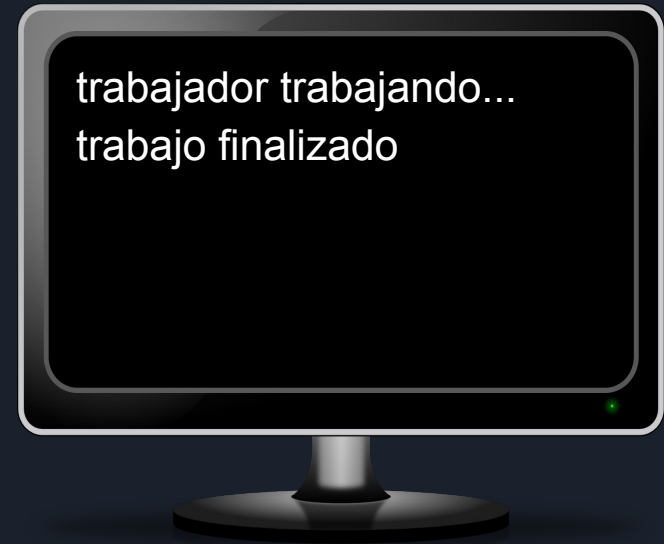
```
        {
```

```
            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-----

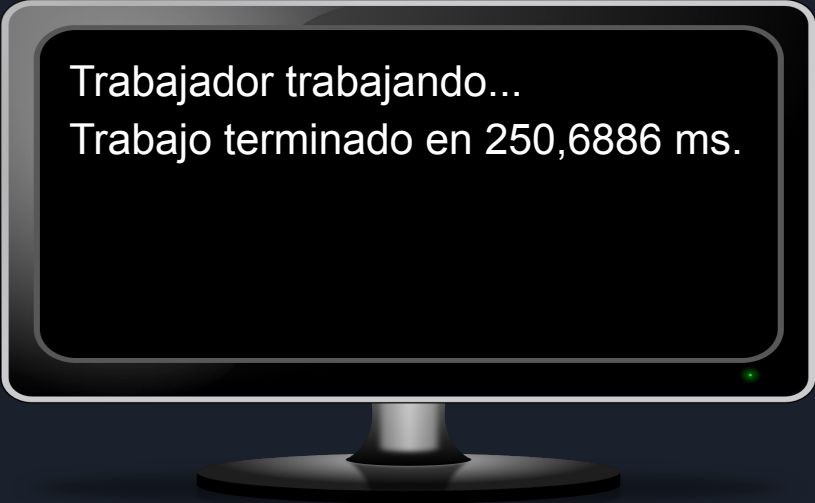
```
class Trabajador
{
    public TrabajoFinalizadoEventHandler? TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("Trabajador trabajando...");
        DateTime tInicial = DateTime.Now;
        for (int i = 1; i < 100_000_000; i++) ; //Pierdo tiempo simulando el trabajo
 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

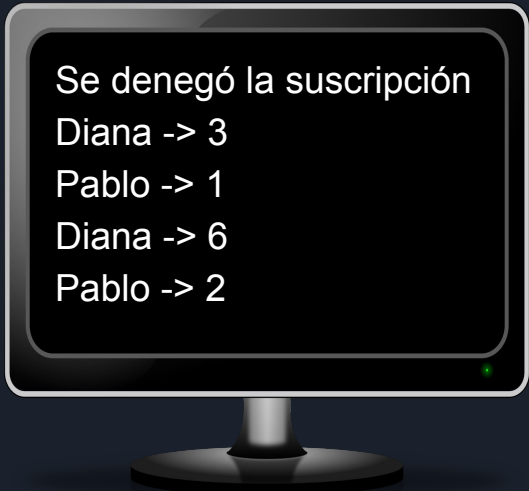
-----Program.cs-----

```
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 del material teórico
complementario

Práctica sobre el material teórico complementario

1) Responder sobre el siguiente código

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

```
AccionInt a1 = (ref int i) => i = i * 2;  
a1 += a1;  
a1 += a1;  
a1 += a1;  
int i = 1;  
a1(ref i);
```

```
-----AccionInt.cs-----
```

```
delegate void AccionInt(ref int i);
```

¿Cuál es el tamaño de la lista de invocación de **a1** y cual es el valor de la variable **i** luego de la invocación **a1(ref i)**?

2) Dado el siguiente código:

```
-----Program.cs-----
Trabajador t1 = new Trabajador();
t1.Trabajando = T1Trabajando;
t1.Trabajar();

void T1Trabajando(object? sender, EventArgs e)
    => Console.WriteLine("Se inició el trabajo");

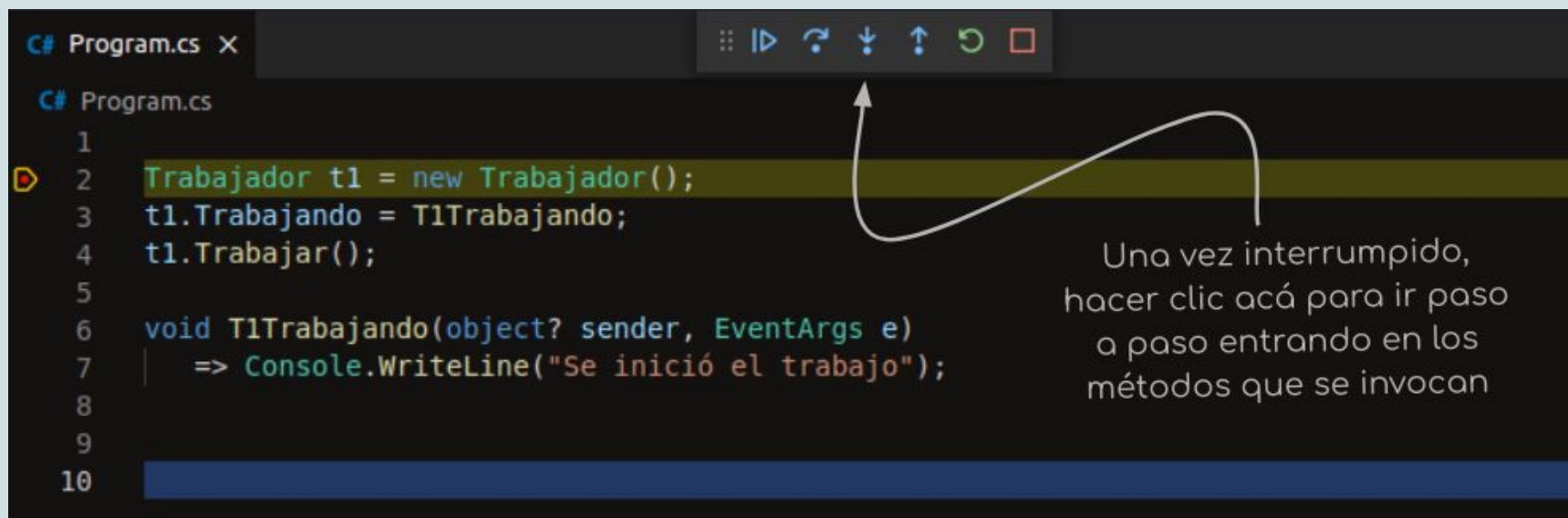
-----Trabajador.cs-----
class Trabajador
{
    public EventHandler? Trabajando; //No es necesario definir un tipo delegado propio
                                     //porque la plataforma provee el tipo EventHandler
                                     //que se adecua a lo que se necesita

    public void Trabajar()
    {
        Trabajando(this, EventArgs.Empty);
        //realiza algún trabajo
        Console.WriteLine("Trabajo concluido");
    }
}
```

a) Ejecutar paso a paso el programa y observar cuidadosamente su funcionamiento. Para ejecutar paso a paso colocar un punto de interrupción (*breakpoint*) en la primera línea ejecutable del método **Main()**



Ejecutar el programa y una vez interrumpido, proseguir paso a paso, en general la tecla asociada para ejecutar paso a paso entrando en los métodos que se invocan es F11, sin embargo también es posible utilizar el botón de la barra que aparece en la parte superior del editor cuando el programa está con la ejecución interrumpida.



```
C# Program.cs X
C# Program.cs
1
2 Trabajador t1 = new Trabajador();
3 t1.Trabajando = T1Trabajando;
4 t1.Trabajar();
5
6 void T1Trabajando(object? sender, EventArgs e)
7 | => Console.WriteLine("Se inició el trabajo");
8
9
10
```

Una vez interrumpido, hacer clic acá para ir paso a paso entrando en los métodos que se invocan

b) ¿Qué salida produce por Consola?

c) Borrar (o comentar) la instrucción `t1.Trabajando = T1Trabajando;` del método `Main` y contestar:

c.1) ¿Cuál es el error que ocurre? ¿Dónde y por qué?

c.2) ¿Cómo se debería implementar el método `Trabajar()` para evitarlo? Resolverlo.

d) Eliminar el método `T1Trabajando` en `Program.cs` y suscribirse al evento con una expresión lambda.

e) Reemplazar el campo público `Trabajando` de la clase `Trabajador`, por un evento público generado por el compilador (event notación abreviada). ¿Qué operador se debe usar en la suscripción?

f) Cambiar en la clase `Trabajador` el evento generado automáticamente por uno implementado de manera explícita con los dos descriptores de acceso y haciendo que, al momento en que alguien se suscriba al evento, se dispare el método `Trabajar()`, haciendo innecesaria la invocación `t1.Trabajar();` en `Program.cs`

3) Analizar el siguiente código

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

```
ContadorDeLineas contador = new ContadorDeLineas();  
contador.Contar();
```

-----ContadorDeLineas.cs-----

```
class ContadorDeLineas  
{  
    private int _cantLineas = 0;  
    public void Contar()  
    {  
        Ingresador _ingresador = new Ingresador();  
        _ingresador.Contador = this;  
        _ingresador.Ingresar();  
        Console.WriteLine($"Cantidad de líneas ingresadas: {_cantLineas}");  
    }  
    public void UnaLineaMas() => _cantLineas++;  
}
```

-----Ingresador.cs-----

```
class Ingresador  
{  
    public ContadorDeLineas? Contador { get; set; }  
    public void Ingresar()  
    {  
        string st = Console.ReadLine()??"";  
        while (st != "")  
        {  
            Contador?.UnaLineaMas();  
            st = Console.ReadLine()??"";  
        }  
    }  
}
```

Existe un alto nivel de acoplamiento entre las clases **ContadorDeLineas** e **Ingresador**, habiendo una referencia circular: un objeto **ContadorDeLineas** posee una referencia a un objeto **Ingresador** y éste último posee una referencia al primero. Esto no es deseable, hace que el código sea difícil de mantener. Eliminar esta referencia circular utilizando un evento, de tal forma que **ContadorDeLineas** posea una referencia a **Ingresador** pero que no ocurra lo contrario.

4) Codificar una clase **Ingresador** con un método público **Ingresar()** que permita al usuario ingresar líneas por la consola hasta que se ingrese la línea con la palabra **"fin"**. Ingresador debe implementar dos eventos. Uno sirve para notificar que se ha ingresado una línea vacía (**""**). El otro para indicar que se ha ingresado un valor numérico (debe comunicar el valor del número ingresado como argumento cuando se genera el evento). A modo de ejemplo observar el siguiente código que hace uso de un objeto **Ingresador**.

```
Ingresador ingresador = new Ingresador();
ingresador.LineaVacíaIngresada += (sender, e) =>
    { Console.WriteLine("Se ingresó una línea en blanco"); };
ingresador.NroIngresado += (sender, e) =>
    { Console.WriteLine($"Se ingresó el número {e.Valor}"); };
ingresador.Ingresar();
```

5) Codificar la clase **Temporizador** con un evento **Tic** que se genera cada cierto intervalo de tiempo medido en milisegundos una vez que el temporizador se haya habilitado. La clase debe contar con dos propiedades: **Intervalo** de tipo **int** y **Habilitado** de tipo **bool**. No se debe permitir establecer la propiedad **Habilitado** en **true** si no existe ninguna suscripción al evento **Tic**. No se debe permitir establecer el valor de **Intervalo** menor a 100. En el lanzamiento del evento, el temporizador debe informar la cantidad de veces que se provocó el evento. Para detener los eventos debe establecerse la propiedad **Habilitado** en **false**. A modo de ejemplo, el siguiente código debe producir la salida indicada.

```
Temporizador t = new Temporizador();
t.Tic += (sender, e) =>
{
    Console.WriteLine(DateTime.Now.ToString("HH:mm:ss") + " ");
    if (e.Tics == 5)
    {
        t.Habilitado = false;
    }
};
t.Intervalo = 2000;
t.Habilitado = true;
```

Salida por consola

```
14:20:50
14:20:52
14:20:54
14:20:56
14:20:58
```