



.NET

Teoría 7

Interfaces



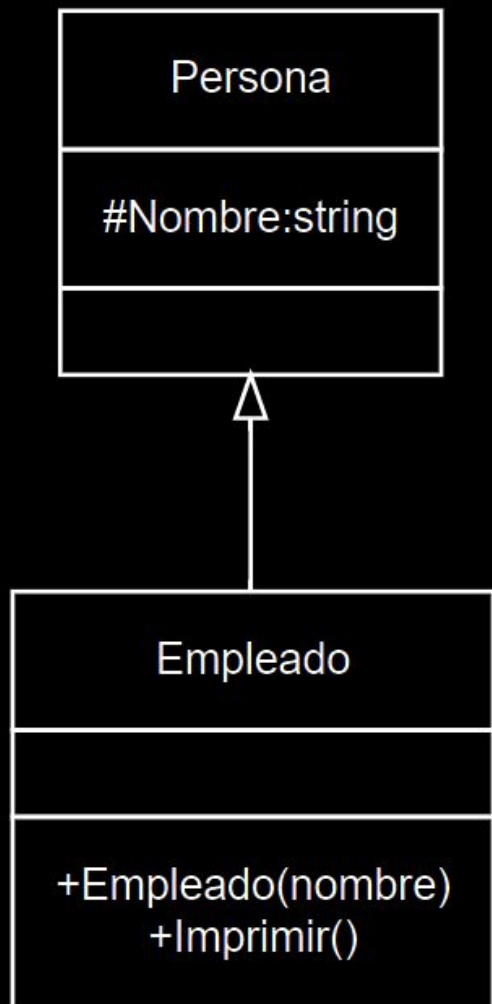
Vamos a presentar el concepto por medio de un ejemplo



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria7`
4. Abrir `Visual Studio Code` sobre este proyecto



Codificar las clases Persona y Empleado



- La clase **Persona**, debe tener un campo protegido de tipo **string** llamado **Nombre**
- La clase **Empleado** debe derivar de **Persona** y contar con un **constructor** que reciba su nombre como parámetro y un método público **Imprimir()** para imprimirse en la consola

```
----- Persona.cs -----
```

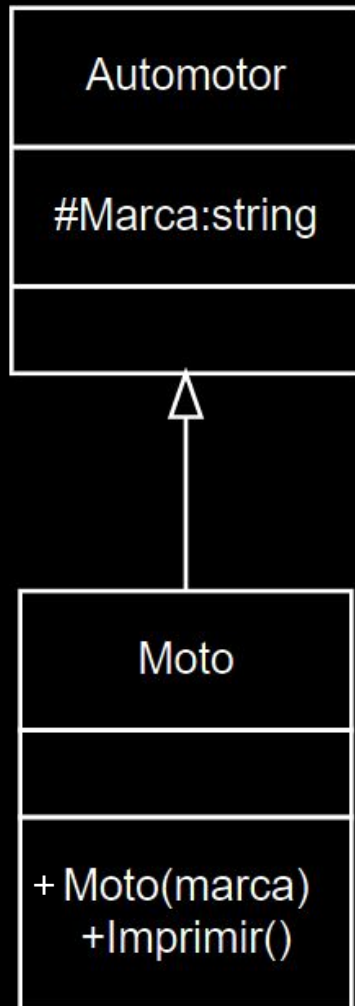
```
namespace Teoria7;  
  
class Persona  
{  
    protected string Nombre = "";  
}
```

```
----- Empleado.cs -----
```

```
namespace Teoria7;  
  
class Empleado : Persona  
{  
    public Empleado(string nombre)  
        => Nombre = nombre;  
    public void Imprimir()  
        => Console.WriteLine($"Soy el empleado {Nombre}");  
}
```



Codificar las clases Automotor y Moto



- La clase **Automotor**, debe tener un campo protegido de tipo **string** llamado **Marca**
- La clase **Moto** debe derivar de **Automotor** y contar con un **constructor** que reciba su marca como parámetro y un método público **Imprimir()** para imprimirse en la consola

```
----- Automotor.cs -----
```

```
namespace Teoria7;  
  
class Automotor  
{  
    protected string Marca = "";  
}
```

```
----- Moto.cs -----
```

```
namespace Teoria7;  
  
class Moto : Automotor  
{  
    public Moto(string marca)  
        => Marca = marca;  
    public void Imprimir()  
        => Console.WriteLine($"Soy una moto {Marca}");  
}
```



Completar Program.cs invocando el método Imprimir de todos los elementos del vector



```
using Teoria7;
```

```
object[] vector = new object[] {  
    new Moto("Zanella"),  
    new Empleado("Juan"),  
    new Moto("Gilera")  
};
```

object es el
ancestro común
más cercano
entre Moto y
Empleado

```
foreach (object o in vector)  
{  
    . . .  
}
```

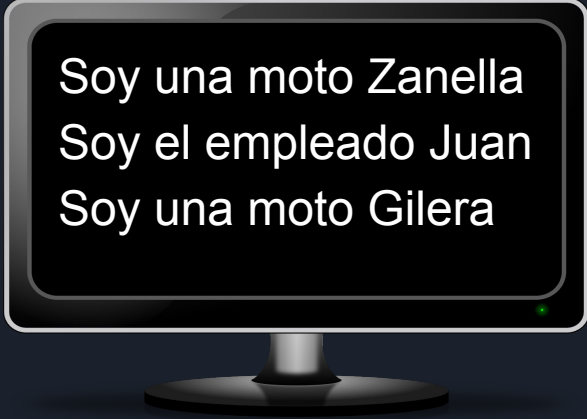
Completar

Posible solución

```
...  
foreach (object o in vector)  
{  
    if (o is Empleado e)  
    {  
        e.Imprimir();  
    }  
    else if (o is Moto m)  
    {  
        m.Imprimir();  
    }  
}  
...
```

equivale a

```
if (o is Empleado)  
{  
    Empleado e = (o as Empleado);  
    e.Imprimir();  
}
```



Soy una moto Zanella
Soy el empleado Juan
Soy una moto Gilera

Solución poco eficiente

```
. . .  
foreach (object o in vector)  
{  
    if (o is Empleado e)  
    {  
        e.Imprimir();  
    }  
    else if (o is Moto m)  
    {  
        m.Imprimir();  
    }  
}  
. . .
```

No hay
polimorfismo



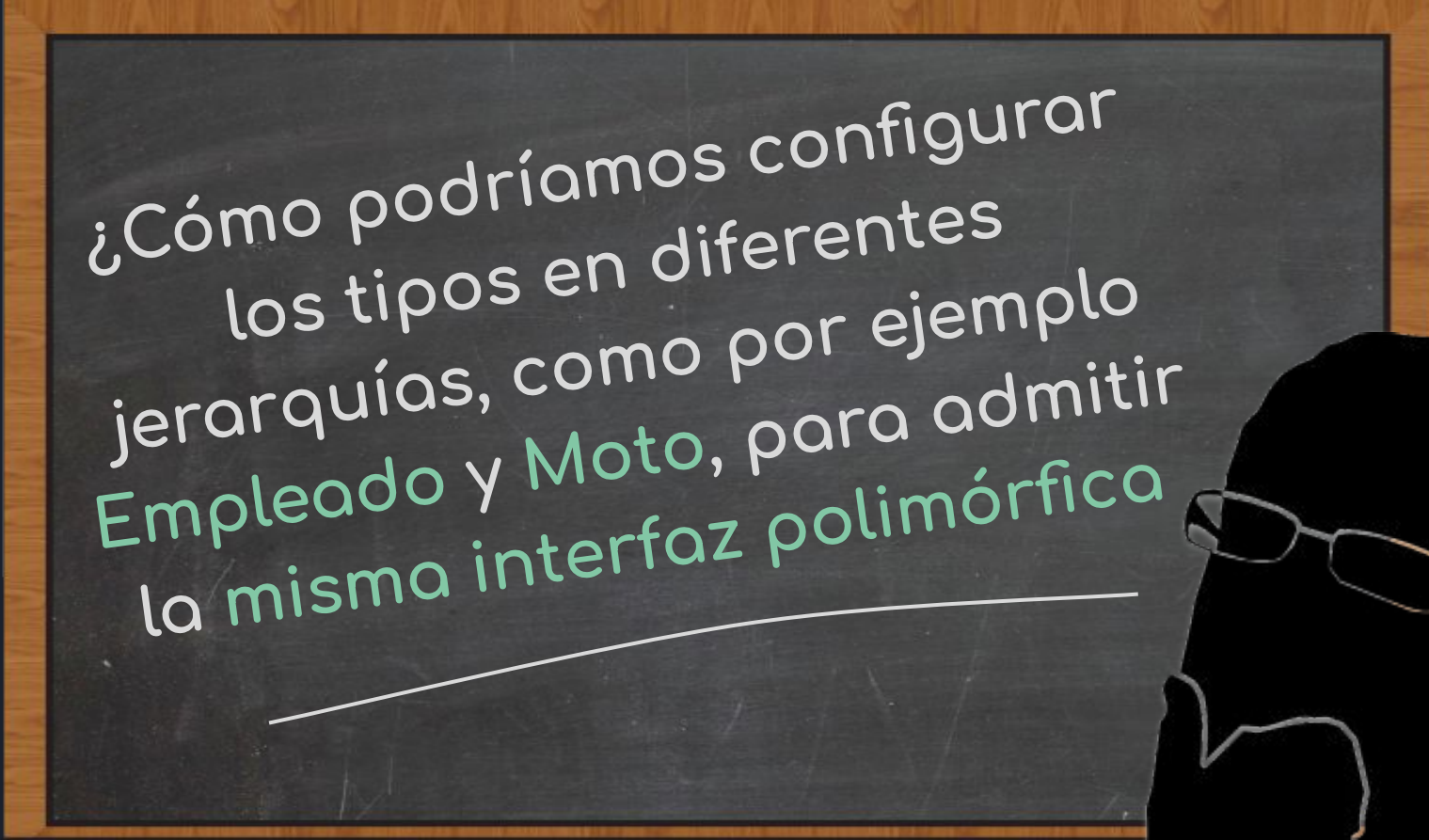
Dificultad para usar polimorfismo

La **interfaz polimórfica** establecida por una **clase base** sólo es aprovechada por los **tipos derivados**

Sin embargo, en sistemas de software más grandes, es común desarrollar **múltiples jerarquías** de clases que no tienen un padre común más allá de **System.Object**.



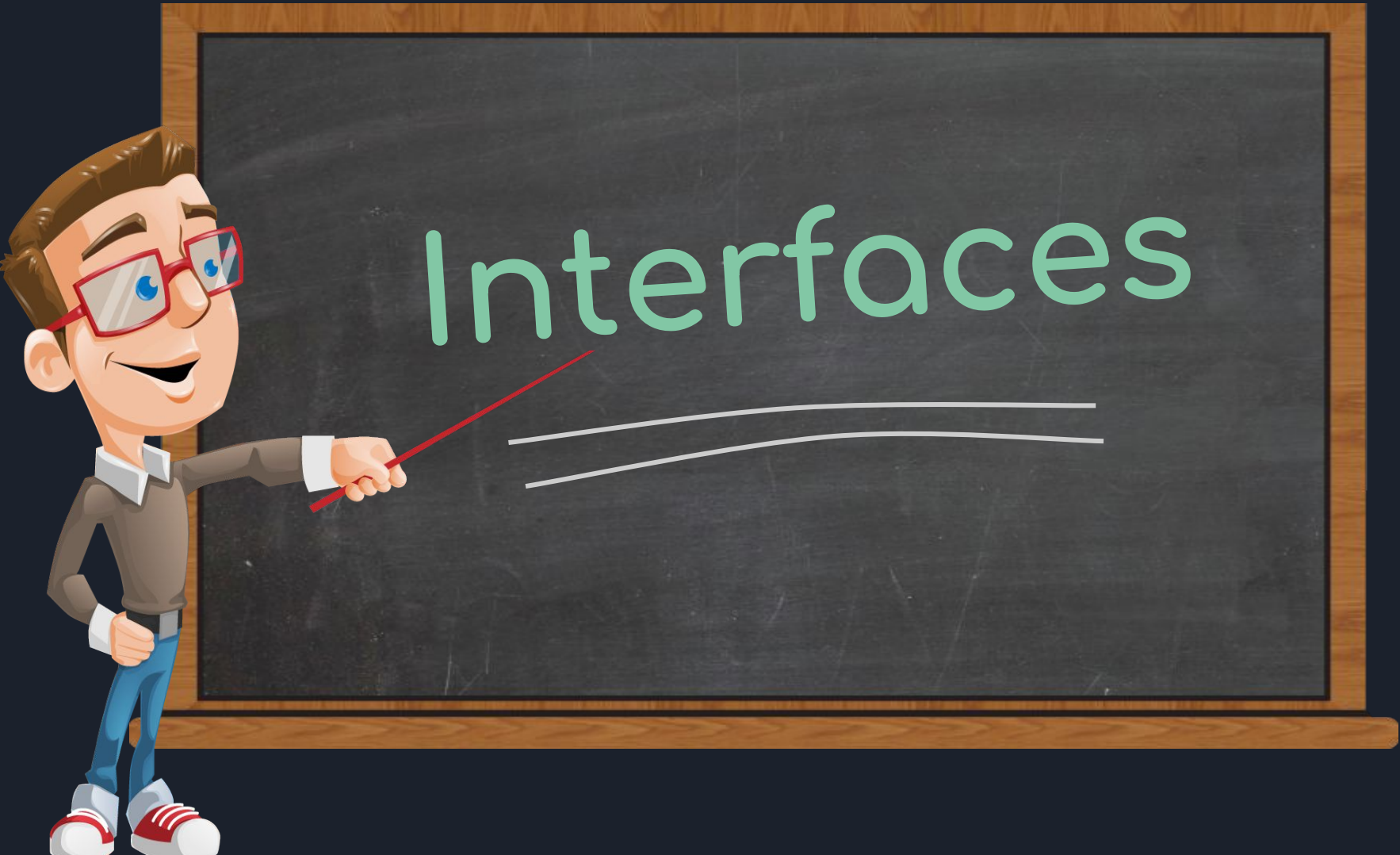
Interrogante



¿Cómo podríamos configurar
los tipos en diferentes
jerarquías, como por ejemplo
Empleado y Moto, para admitir
la misma interfaz polimórfica



Respuesta



¿ Qué es una Interfaz ?

- Es un **tipo referencia** que especifica un conjunto de **funciones sin implementarlas**.
- Pueden especificar **métodos, propiedades, indizadores y eventos** de instancia, sin implementación (o con implementación predeterminada a partir de **c# 8.0**).
- En lugar del código que los implementa llevan un punto y coma (;)
- Por convención comienzan con la letra **I** (i latina mayúscula)

¿ Qué es una Interfaz ?

- A partir de C# 8.0, una interfaz puede definir una implementación predeterminada de miembros.
- A partir de C# 8.0, una interfaz puede definir miembros estáticos (con implementación)
- Una interfaz no puede contener campos de instancia, constructores de instancia ni finalizadores

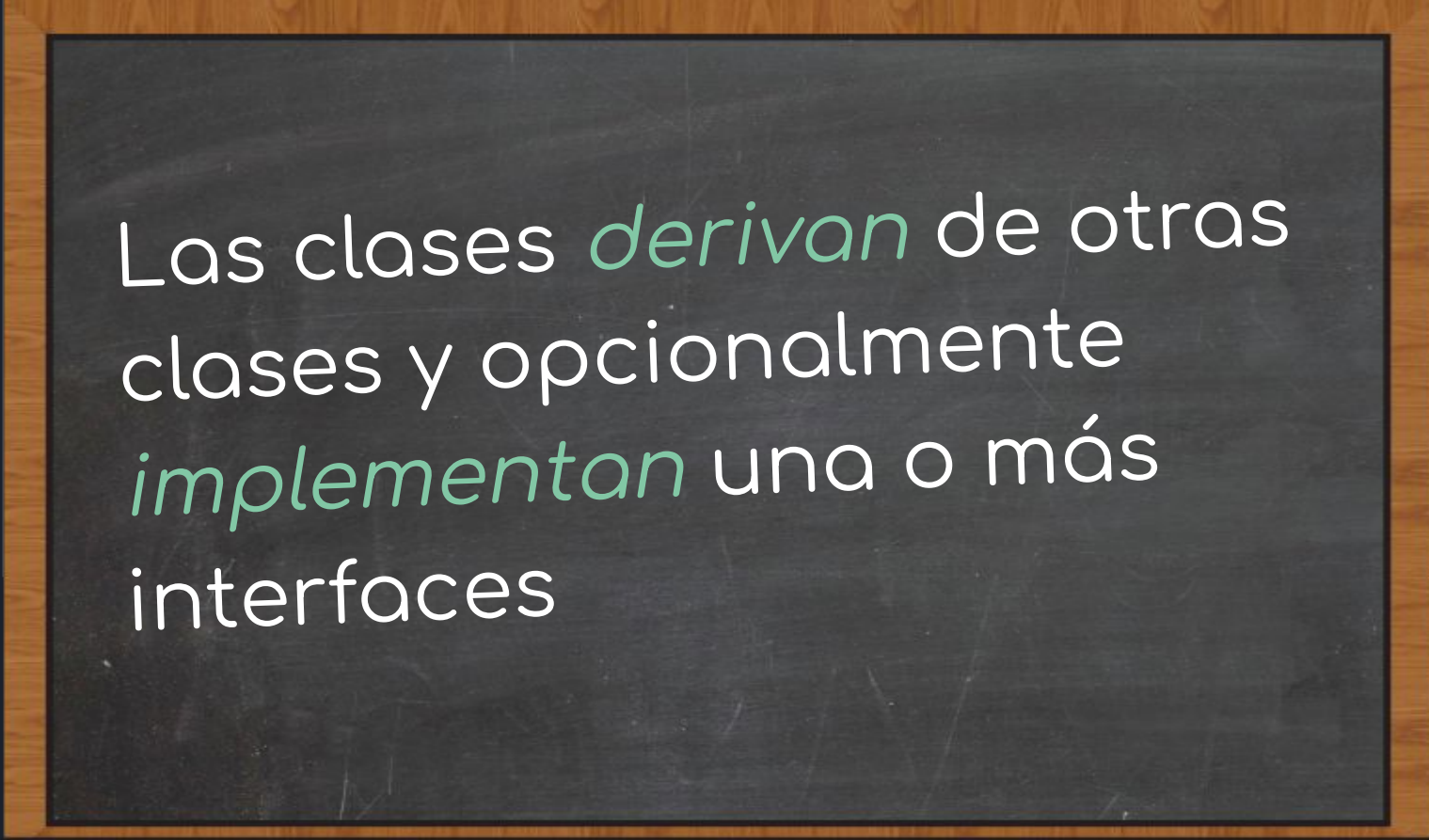
Declarando una interfaz

Los miembros de las interfaces **son públicos por defecto**. En versiones del lenguaje anteriores a C# 8.0 no se permite utilizar modificadores de acceso (ni siquiera **public**)

```
public interface IMiInterface
{
    public void UnMetodo();
}
```

Atención, no usar modificadores si se está utilizando una versión anterior a C# 8.0

Implementación de interfaces



Las clases *derivan* de otras
clases y opcionalmente
implementan una o más
interfaces

Implementación de interfaces

Si una clase implementa una interfaz debe implementar todos los miembros de la interfaz que no tienen implementación predeterminada.

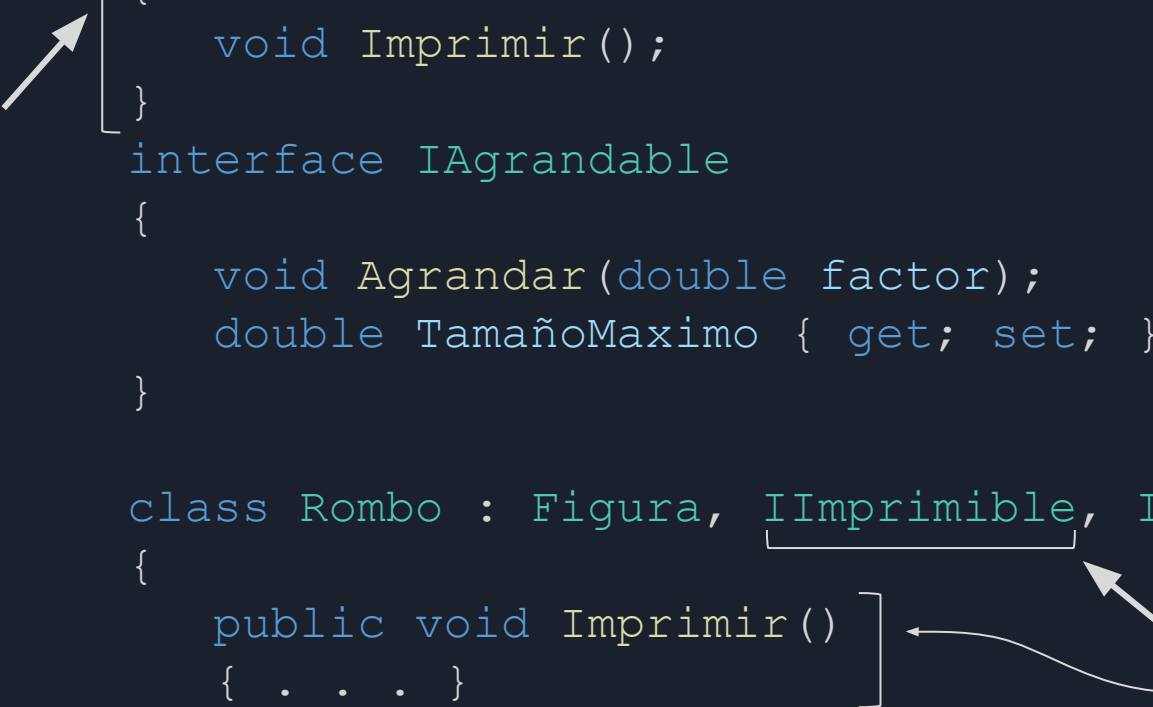
Si una clase deriva de otra clase y además implementa algunas interfaces, la clase debe ser la primera en la lista después de los dos puntos

```
class Rombo : Figura, IImprimible, IAgrandable
{
    . . .
}
```

Clase base

Interfaces

Interfaces - Implementación de interfaces



```
interface IImprimible
{
    void Imprimir();
}
interface IAgrandable
{
    void Agrandar(double factor);
    double TamañoMaximo { get; set; }
}
```


```
class Rombo : Figura, IImprimible, IAgrandable
{
    public void Imprimir()
    { . . . }
    public void Agrandar(double factor)
    { . . . }
    public double TamañoMaximo
    {
        get { . . . }
        set { . . . }
    }
}
```

Obligado a
implementarlo

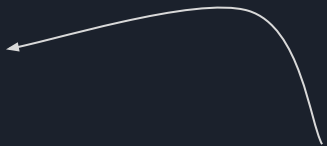

Interfaces - Implementación de interfaces

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

```
interface IAgrandable
{
    void Agrandar(double factor);
    double TamañoMaximo { get; set; }
}
```



```
class Rombo : Figura, IImprimible, IAgrandable
{
    public void Imprimir()
    { . . . }
    public void Agrandar(double factor)
    { . . . }
    public double TamañoMaximo
    {
        get { . . . }
        set { . . . }
    }
}
```



Obligado a
implementarlos

Utilización de interfaces

Es posible definir y utilizar variables de tipo interface.
Por ejemplo:

```
Rombo r1 = new Rombo();  
Figura r2 = new Rombo();  
IAgrandable r3 = new Rombo();  
IImprimible r4 = new Rombo();
```

Las siguientes son sentencias son válidas

```
r3.TamañoMaximo = 100;  
r3.Agrandar(1.2);  
r4.Imprimir();  
(r3 as IImprimible).Imprimir();
```

Utilización de interfaces

Las interfaces **son tipos de referencia**, por lo tanto es posible utilizar el operador **as** (ya lo vimos). Es habitual combinar su uso con el del operador **is** de la siguiente manera:

```
. . .  
object o;  
. . .  
if (o is IImprimible)  
{  
    (o as IImprimible).Imprimir();  
}  
. . .
```

```
. . .  
object o;  
. . .  
if (o is IImprimible imp)  
{  
    imp.Imprimir();  
}  
. . .
```

↑
facilidad
incorporada en la
versión 7.0 de C#

Utilización de interfaces

No es posible crear una instancia de una interface

```
IImprimible imp = new IImprimible();
```



No está permitido



Utilización de interfaces

Sí se puede hacer esto

```
IImprimible[] vector = new IImprimible[10];
```

Acá no instanciamos ningún
objeto `IImprimible`.

Los elementos que agreguemos al
vector (inicialmente todos null)
tendrán que implementar la
interface `IImprimible`.



Utilización de interfaces

También es posible utilizar tipos Interfaz para las propiedades, indizadores y métodos (argumentos y valor de retorno)

```
IImprimible Elemento {get;set;}  
IImprimible this [int index]...  
void EstablecerElemento(IImprimible e)...  
IImprimible ObtenerElemento()...
```





Resolver el ejercicio inicial de manera polimórfica



----- IImprimible.cs -----

```
namespace Teoria7;
```

```
interface IImprimible
```

```
{
```

```
    void Imprimir();
```

```
}
```



----- Automotor.cs -----

```
class Moto : Automotor, IImprimible
```

```
{
```

```
    . . .
```

```
}
```



----- Automotor.cs -----

```
class Empleado : Persona, IImprimible
```

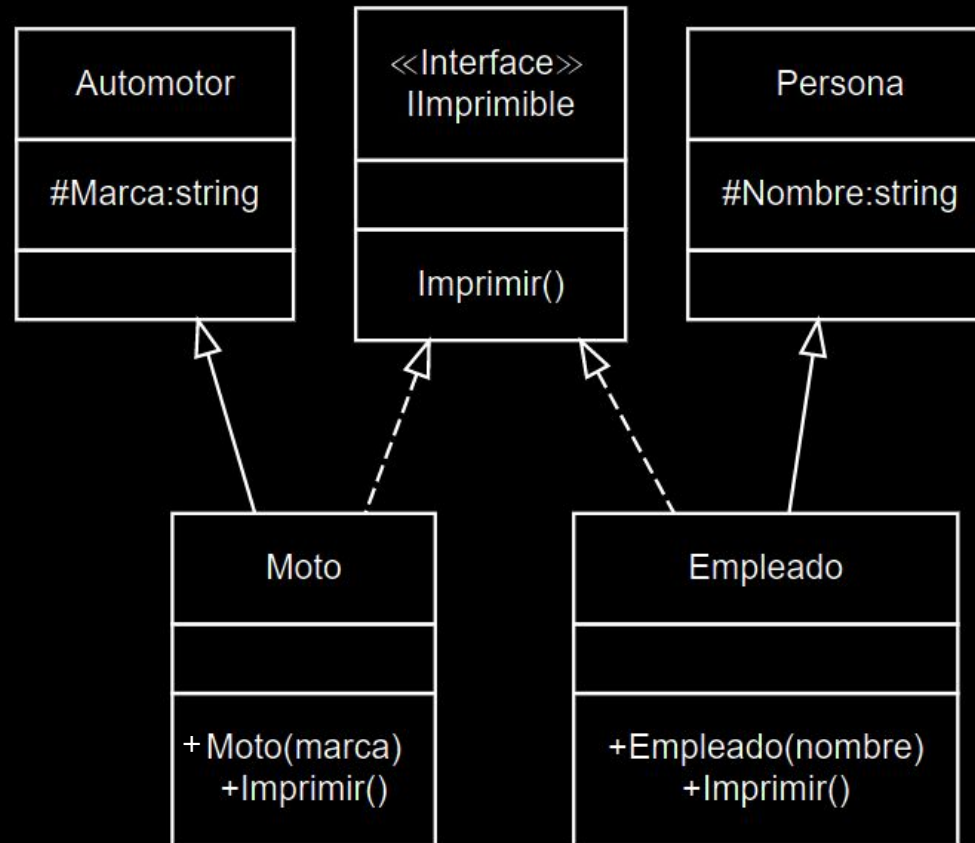
```
{
```

```
    . . .
```

```
}
```



Resolución del ejercicio inicial utilizando interfaces





Codificar Program.cs para obtener una Solución polimórfica



```
using Teoria7;
```

```
object[] vector = new object[] {  
    new Moto("Zanella"),  
    new Empleado("Juan"),  
    new Moto("Gilera")  
};
```

```
foreach (IImprimible imp in vector)  
{  
    imp.Imprimir();  
}
```

Acá hay una conversión de
tipo de `object` a `IImprimible`

Solución alternativa (mejor)

```
using Teoria7;
```

```
IImprimible[] vector = new IImprimible[] {  
    new Moto("Zanella"),  
    new Empleado("Juan"),  
    new Moto("Gilera")  
};
```

```
for (int i = 0; i < vector.Length; i++)  
{  
    vector[i].Imprimir();  
}
```

Vector de elementos `IImprimible`
Esta solución es más segura, se aprovecha la verificación de tipo que hace el compilador

Funciona sin necesidad de conversión alguna porque los elementos son `IImprimible`


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");
}
```

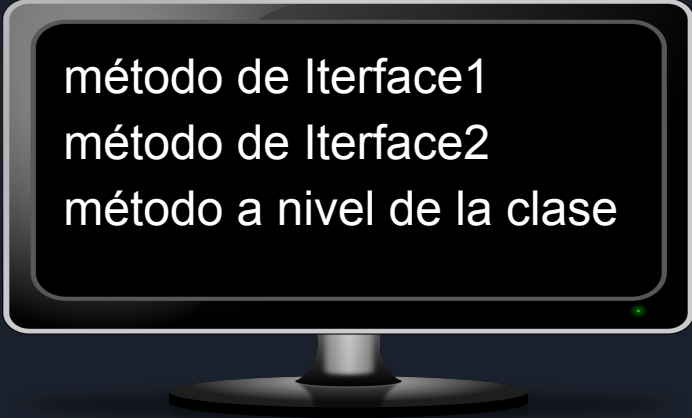


IMPORTANTE:

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

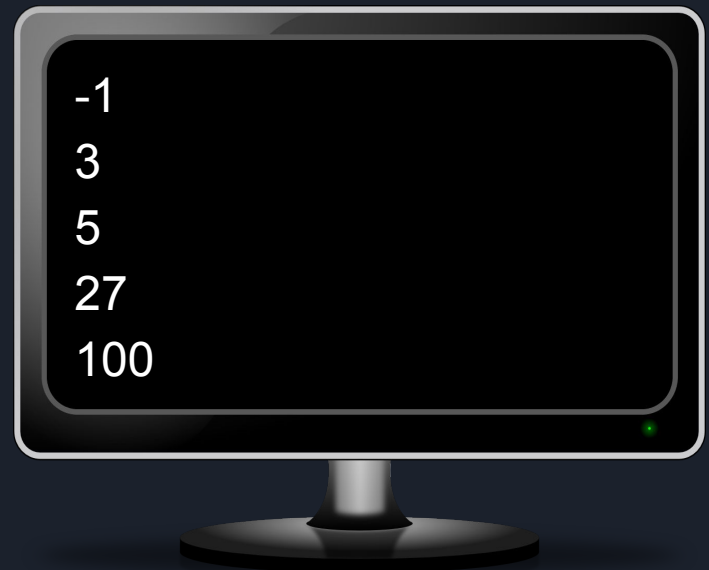
- una implementación a nivel de clase
- una implementación explícita de interface
- Ambas, una implementación explícita de interface y una implementación a nivel de clase

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**





Ordenamiento - Ejemplo 2

Codificar Program.cs de la siguiente manera



```
using Teoria7;

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

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**

```
C# Program.cs
1  using Teoria7;
2
3  var vector = new Empleado[] {
4      |         |         |         |         |         |         |         |
5      |         |         |         |         |         |         |         |
6      |         |         |         |         |         |         |         |
7      |         |         |         |         |         |         |         |
8  Array.Sort(vector);
```

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 : Persona, IImprimible, 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;
    }
}
```

. . .

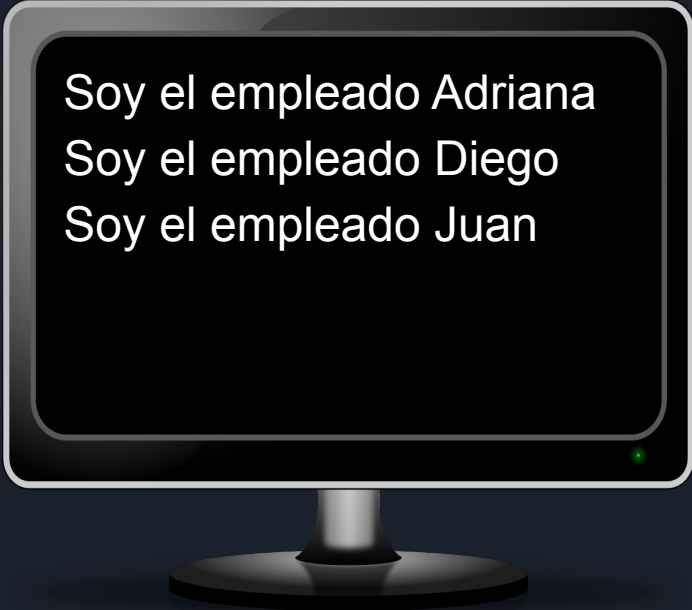
Ordenamiento Ejemplo 2



```
using Teoria7;

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 : Persona, IImprimible, 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 Teoria7;

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 Teoria7;
```

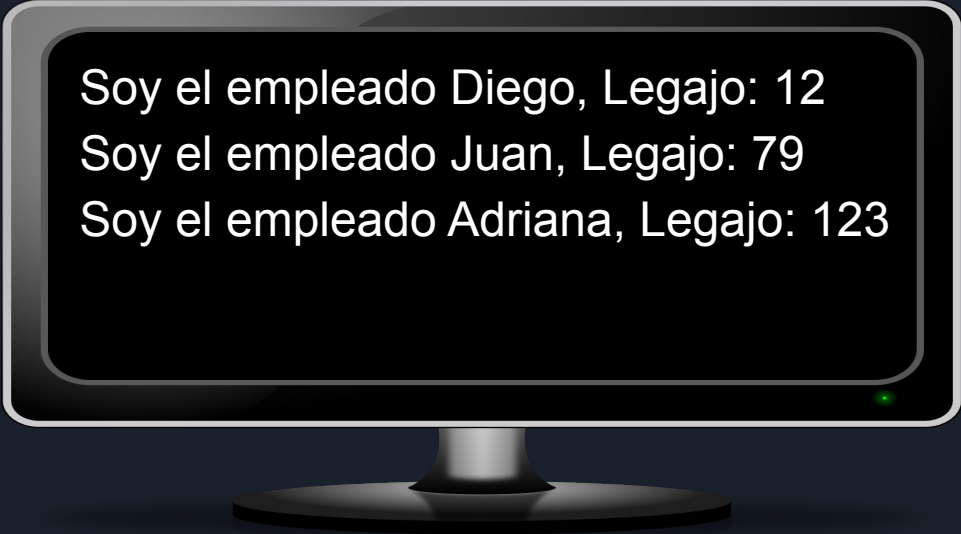
```
var vector = new Empleado[] {  
    new Empleado("Juan") {Legajo=79},  
    new Empleado("Adriana") {Legajo=123},  
    new Empleado("Diego") {Legajo=12}  
};
```

```
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;

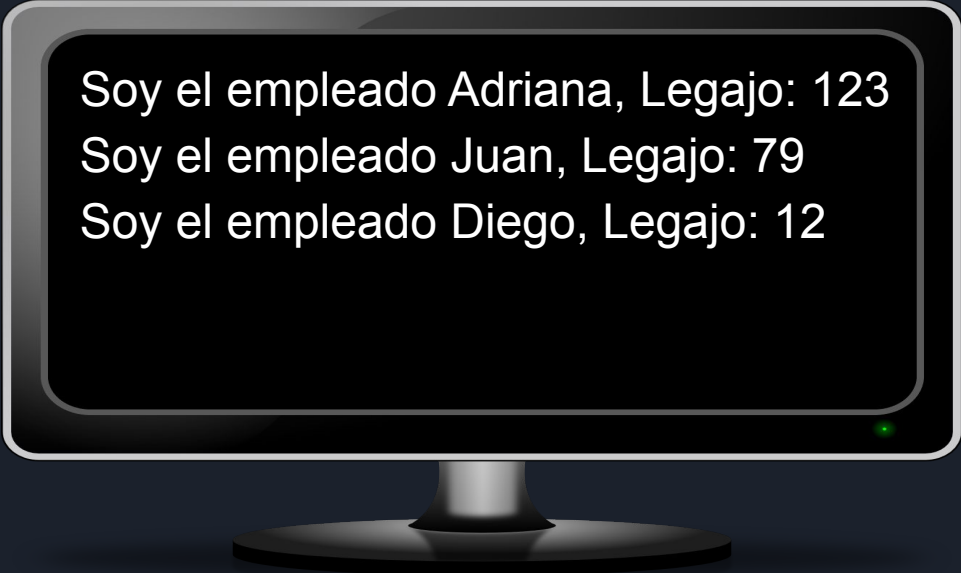
    public int Compare(object? x, object? y)
    {
        int result = 1;
        if (x is Empleado && y is Empleado)
        {
            int legajo1 = ((Empleado)x).Legajo;
            int legajo2 = ((Empleado)y).Legajo;
            result = legajo1.CompareTo(legajo2);
        }
        if (Descendente)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

Modificando
`ComparadorPorLegajo` para
permitir ordenar ascendente o
descendentemente

```
using Teoria7;

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`

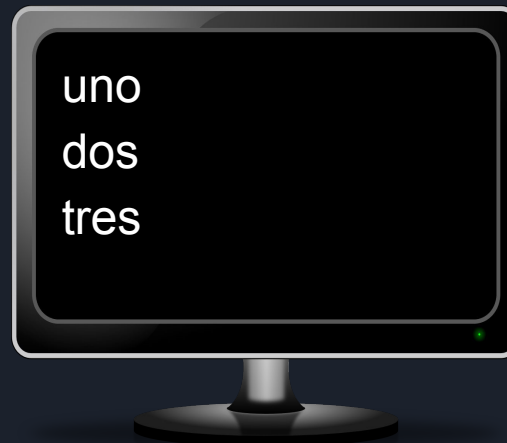
Uso de la instrucción foreach Ejemplo 1

. . .

```
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 Teoria7;

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 Teoria7;
```

```
Pyme miPyme = new Pyme(new Empleado("Juan") { Legajo = 79 },  
                        new Empleado("Adriana") { Legajo = 123 },  
                        new Empleado("Diego") { Legajo = 12 });
```

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

Error de compilación

```
using Teoria7;
```

```
Pyme miPyme = new Pyme(new Empleado("Juan") { Legajo = 79 },  
                        new Empleado("Adriana") { Legajo = 123 },  
                        new Empleado("Diego") { Legajo = 12 });
```

```
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 Teoria7;
```

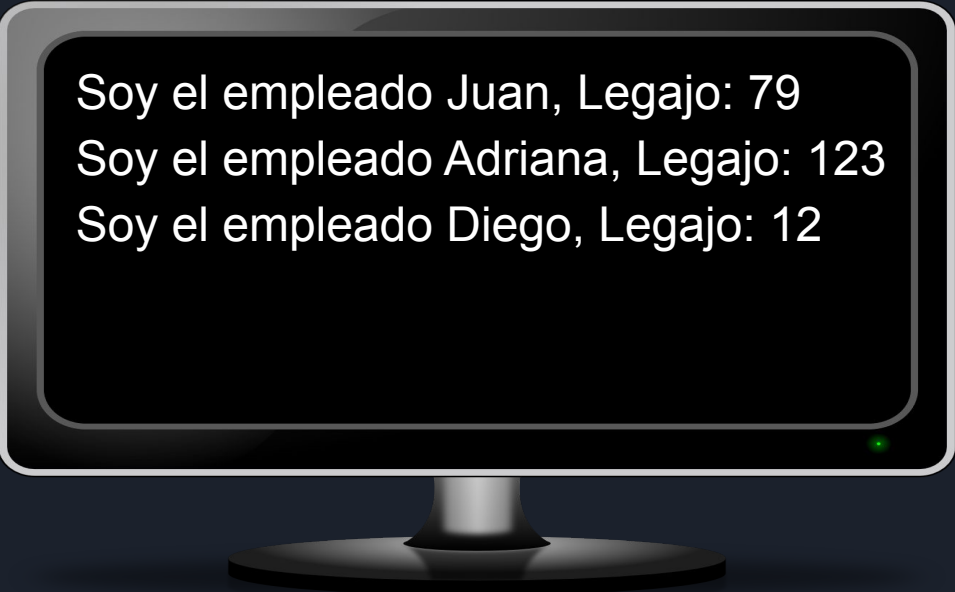
```
class Pyme: IEnumerable  
{  
    Empleado[] empleados = new Empleado[3];  
    public Pyme(Empleado e1, Empleado e2, Empleado e3)  
    {  
        empleados[0] = e1;  
        empleados[1] = e2;  
        empleados[2] = e3;  
    }  
}
```

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

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

```
using Teoria7;  
  
Pyme miPyme = new Pyme(new Empleado("Juan") { Legajo = 79 },  
                        new Empleado("Adriana") { Legajo = 123 },  
                        new Empleado("Diego") { Legajo = 12 });  
  
foreach (Empleado e in miPyme)  
{  
    e.Imprimir();  
}
```

Solucionado !



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

¿ 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();

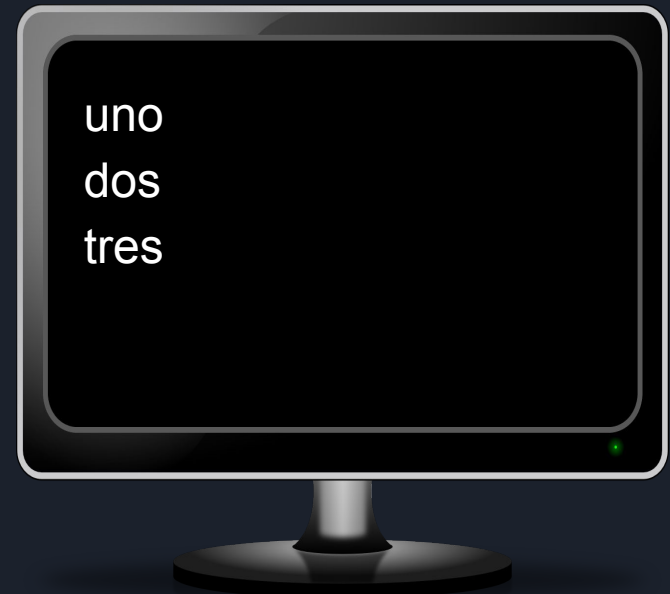
        // Sets the enumerator before the first element
        void Reset();
    }
}
```


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);
}
```



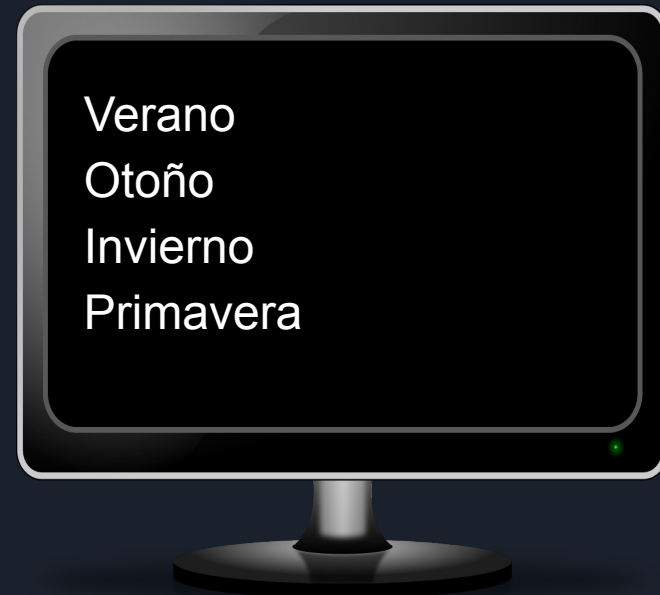
Verano
Otoño
Invierno
Primavera

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

```
using System.Collections;
```

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

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

```
{
```

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

```
}
```

Current es de tipo
object

```
IEnumerator colores()
```

```
{
```

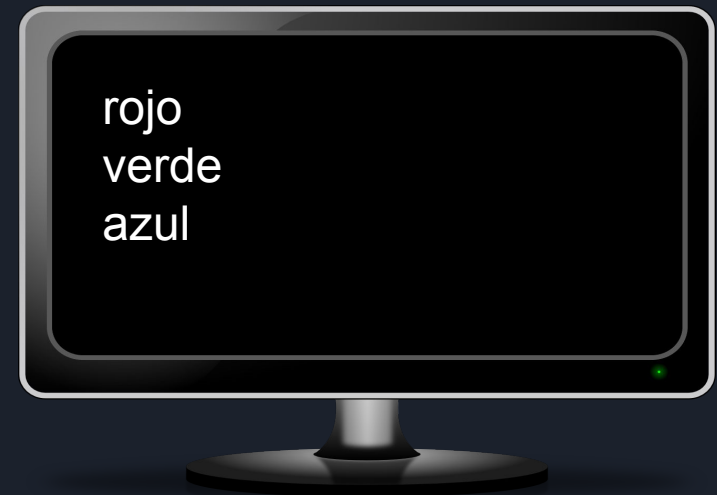
```
    yield return "rojo";
```

```
    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
```



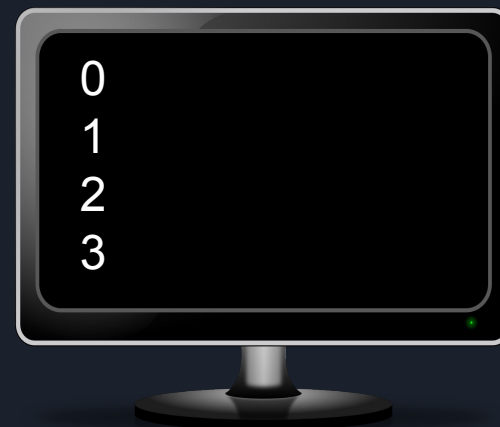
Cuidado!
Un enumerador
generado con un
iterador no implementa
el método **Reset()**

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);
}

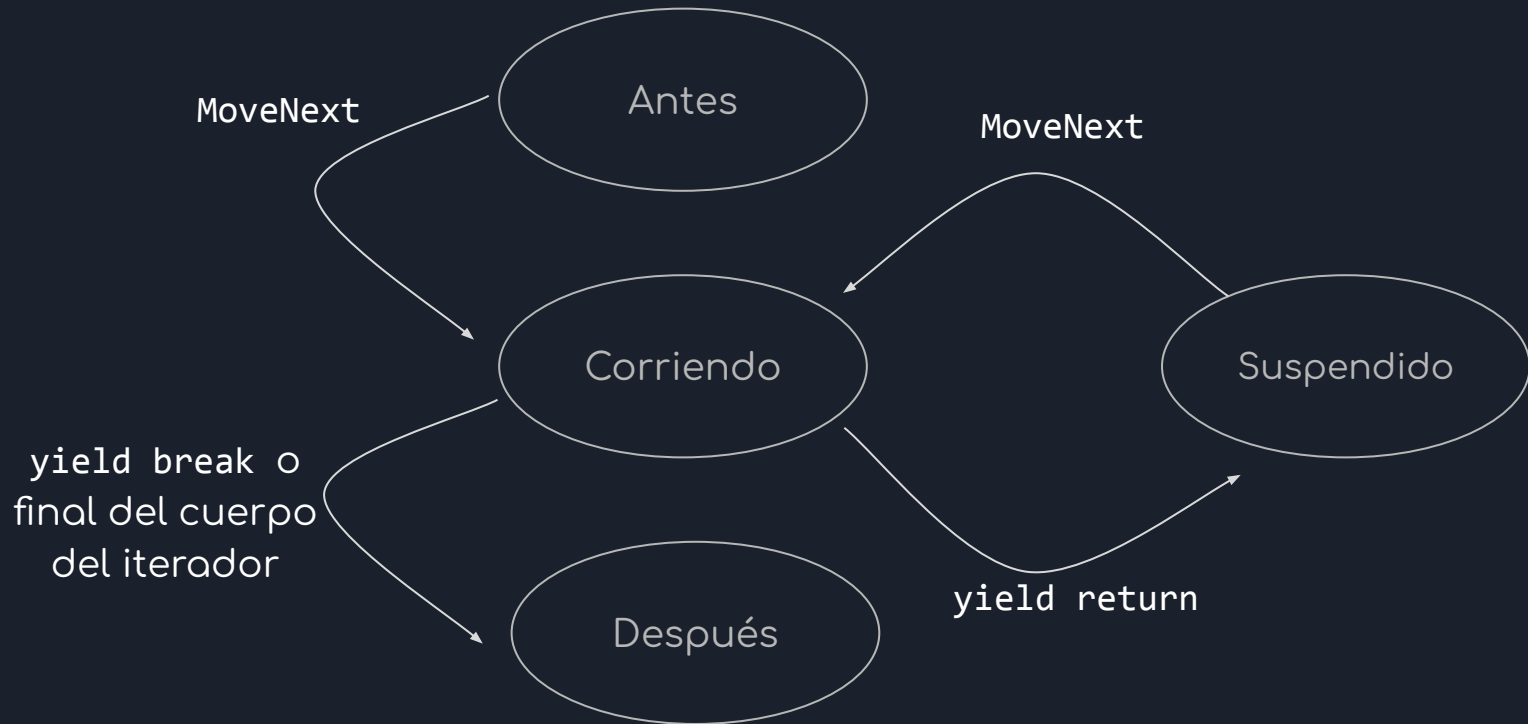
IEnumerable PoderesEstado()
{
    yield return "Ejecutivo";
    yield return "Legislativo";
    yield return "Judicial";
}
```

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;
    }
}
```

```
static 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

Notas complementarias



File System Espacio de nombres System.IO

La BCL incluye todo un espacio de nombres llamado **System.IO** especialmente orientado al trabajo con archivos. Entre las clases más utilizadas de este espacio están:

- Path
- Directory
- DirectoryInfo
- File
- FileInfo

La clase Path

La clase `Path` incluye un conjunto de miembros estáticos diseñados para realizar cómodamente las operaciones más frecuentes relacionadas con rutas y nombres de archivos.


Con los campos públicos `VolumeSeparatorChar`, `DirectorySeparatorChar`, `AltDirectorySeparatorChar` y `PathSeparator`, se obtiene el carácter específico de la plataforma que se utiliza para separar unidades, carpetas y archivos, y el separador de múltiples rutas.

- Con Windows, estos caracteres son `.`, `\`, `/` y `:`
- Con Linux, estos caracteres son `.`, `/`, `/` y `:`

Ejemplo:

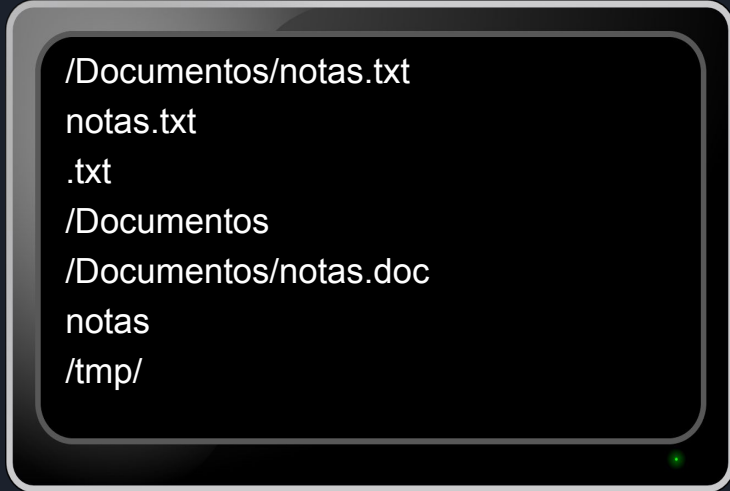
```
string archivo = "/Documentos/notas.txt";  
Console.WriteLine(Path.GetFullPath(archivo));  
Console.WriteLine(Path.GetFileName(archivo));  
Console.WriteLine(Path.GetExtension(archivo));  
Console.WriteLine(Path.GetDirectoryName(archivo));  
Console.WriteLine(Path.ChangeExtension(archivo, "doc"));  
Console.WriteLine(Path.GetFileNameWithoutExtension(archivo));  
Console.WriteLine(Path.GetTempPath());
```

Muchos métodos
sólo procesan
strings



```
C:\Documentos\notas.txt  
notas.txt  
.txt  
\Documentos  
/Documentos/notas.doc  
notas  
C:\Users\Leo\AppData\Local\Temp\
```

En Windows



```
/Documentos/notas.txt  
notas.txt  
.txt  
/Documentos  
/Documentos/notas.doc  
notas  
/tmp/
```

En Linux

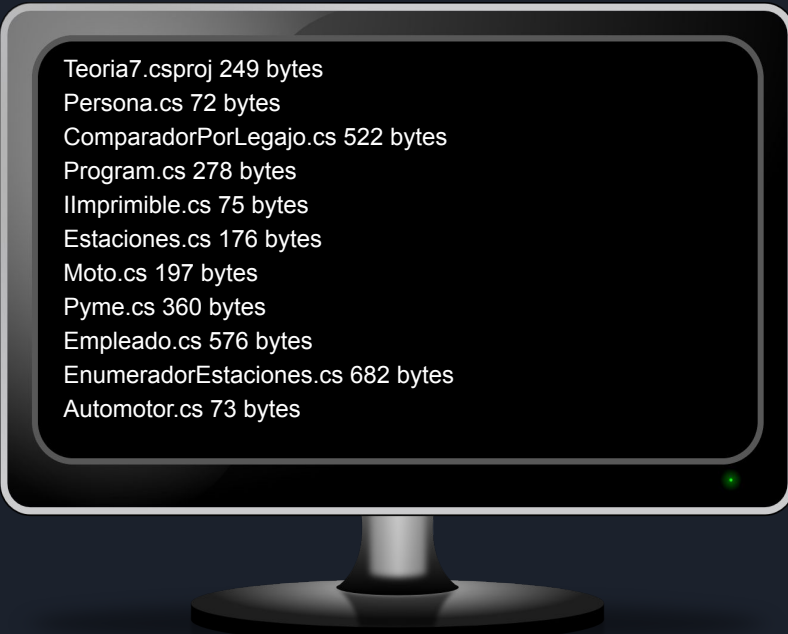


Las clases DirectoryInfo, FileInfo, Directory y File

- Para trabajar con archivos se utilizan objetos de la clase `FileInfo` y para trabajar con directorios objetos de la clase `DirectoryInfo`.
- Las clases `File` y `Directory` que sólo tienen métodos estáticos (al igual que `Path`) son útiles para realizar tareas sencillas. No requieren la creación de ningún objeto pero son menos poderosas y menos eficientes.

DirectoryInfo y FileInfo ejemplo

```
string stDir = Environment.CurrentDirectory;
DirectoryInfo dirInfo = new DirectoryInfo(stDir);
FileInfo[] archivos = dirInfo.GetFiles();
foreach (FileInfo archivo in archivos)
{
    string st = $"{archivo.Name} {archivo.Length} bytes";
    Console.WriteLine(st);
}
```



Teoria7.csproj 249 bytes
Persona.cs 72 bytes
ComparadorPorLegajo.cs 522 bytes
Program.cs 278 bytes
IImprimible.cs 75 bytes
Estaciones.cs 176 bytes
Moto.cs 197 bytes
Pyme.cs 360 bytes
Empleado.cs 576 bytes
EnumeradorEstaciones.cs 682 bytes
Automotor.cs 73 bytes

Archivos de texto

El trabajo con archivos en .NET está ligado al concepto de **stream** o flujo de datos, que consiste en tratar su contenido como una **secuencia ordenada de datos**.

El concepto de **stream** es aplicable también a otros tipos de almacenes de información tales como **conexiones de red** o **buffers en memoria**.

La **BCL** proporciona las clases **StreamReader** y **StreamWriter**. Los objetos de estas clases facilitan la lectura y escritura de **archivos de textos**.



StreamReader

- Para facilitar la **lectura de flujos de texto** **StreamReader** ofrece una familia de métodos que permiten leer sus caracteres de diferentes formas:
- **De uno en uno**: El método **int Read()** devuelve el próximo carácter del flujo. Tras cada lectura la posición actual en el flujo se mueve un carácter hacia delante.



StreamReader

- **Por líneas:** El método `string ReadLine()` devuelve la siguiente línea del flujo (y avanza la posición en el flujo). Una línea de texto es cualquier secuencia de caracteres terminada en `'\n'`, `'\r'` ó `"\r\n"`, aunque la cadena que devuelve no incluye dichos caracteres.
- **Por completo:** `string ReadToEnd()`, que nos devuelve una cadena con todo el texto que hubiese desde la posición actual hasta el final (y avanza hasta el final del flujo).

StreamWriter

`StreamWriter` ofrece métodos que permiten:

- **Escribir cadenas de texto:** El método `Write()` escribe cualquier cadena de texto en el destino que tenga asociado. Pueden utilizarse formatos compuestos.
- **Escribir líneas de texto:** El método `WriteLine()` funciona igual que `Write()` pero añade un indicador de fin de línea. Pueden utilizarse formatos compuestos




StreamWriter

- Dado que el indicador de fin de línea depende de cada sistema operativo, `StreamWriter` dispone de una propiedad string `NewLine` mediante la que puede configurarse este indicador. Su valor por defecto es el “\r\n” en `Windows` y “\n” en `Linux`.

Ejemplo 1 – leyendo y escribiendo archivo de texto fuente en destino

```
StreamReader sr = new StreamReader("fuente.txt");
StreamWriter sw = new StreamWriter("destino.txt");
string? linea;
while (!sr.EndOfStream)
{
    linea = sr.ReadLine();
    sw.WriteLine(linea);
}
sr.Close(); sw.Close();
```

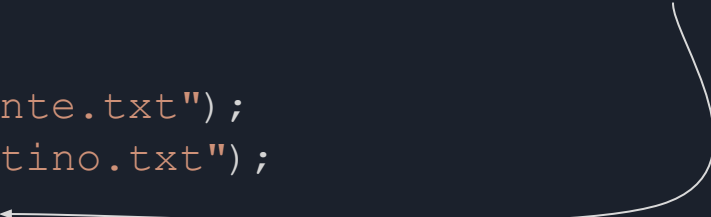


El método `close()` libera los recursos de manera explícita, invocando un método `Dispose()`

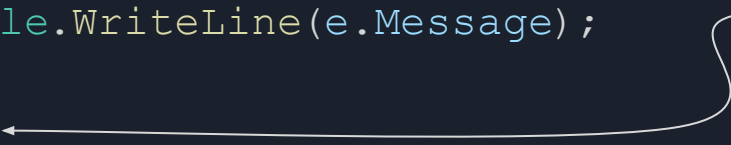
Ejemplo 2 - Manejando excepciones

```
StreamReader? sr = null;
StreamWriter? sw = null;
try
{
    sr = new StreamReader("fuente.txt");
    sw = new StreamWriter("destino.txt");
    sw.Write(sr.ReadToEnd());
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
    sr?.Dispose();
    sw?.Dispose();
}
```

Esta línea hace todo el trabajo



Es recomendable proveer manejo de excepciones y liberar los recursos en un bloque **finally**



Se puede usar **Dispose()** en lugar de **Close()**



Interrogante

¿ Cómo podemos saber si es necesario liberar recursos explícitamente cuando dejamos de usar un objeto de la BCL en nuestro código ?



Respuesta

Fácil, verificar si
implementa la
interface
IDisposable



Interface IDisposable

- En C #, la alternativa recomendada al uso de finalizadores, es implementar la interfaz `System.IDisposable` que posee un único método.

```
public interface IDisposable
{
    void Dispose();
}
```

- `IDisposable` define un mecanismo **determinista** para liberar recursos no administrados y **evita** los **problemas** relacionados con el **recolector de basura** inherentes a los finalizadores.

Interface IDisposable

- Cuando se termina de usar un objeto que implementa `IDisposable`, se debe invocar el método `Dispose()` del objeto. Hay dos maneras de hacerlo:
 - Mediante un bloque `try/finally` como lo hicimos en el último ejemplo de manejo de archivos de texto.
 - Mediante la instrucción `using` (no es la directiva `using` que venimos usando para hacer referencia a los `espacios de nombres`)

Instrucción using

```
using(TipoDisposable recurso = new TipoDisposable(...))  
{  
    bloque de sentencias  
}
```

es equivalente a

```
TipoDisposable recurso = new TipoDisposable(...)  
try {  
    bloque de sentencias  
} finally {  
    if (recurso != null) recurso.Dispose();  
}
```

Cuidado!
No hay
bloque catch

Instrucción using

- En una instrucción `using` se pueden instanciar más de un objeto del mismo tipo, por ejemplo:

```
using (StreamReader f1 = new StreamReader("file1.txt"),  
      f2 = new StreamReader("file2.txt"))  
{  
    ...  
}
```

- Si se trata de distintos tipos los `using` se pueden anidar, como se observa en el ejemplo de la diapositiva siguiente

Instrucción using - Ejemplo

```
try
{
    using (StreamReader sr = new StreamReader("fuente.txt"))
    {
        using (StreamWriter sw = new StreamWriter("destino.txt"))
        {
            sw.Write(sr.ReadToEnd());
        }
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Sin embargo, C# 8.0 introdujo las declaraciones using que proveen una sintaxis más limpia

Declaración using

- Una declaración `using` es una declaración de variable precedida de la `palabra clave using`. El compilador invocará el `Dispose()` sobre la variable declarada al final del ámbito de inclusión. El ejemplo de la diapositiva anterior puede escribirse de la siguiente manera:

```
try
{
    using StreamReader sr = new StreamReader("fuente.txt");
    using StreamWriter sw = new StreamWriter("destino.txt");
    sw.Write(sr.ReadToEnd());
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Fin