



.NET

Teoría 7

Interfaces



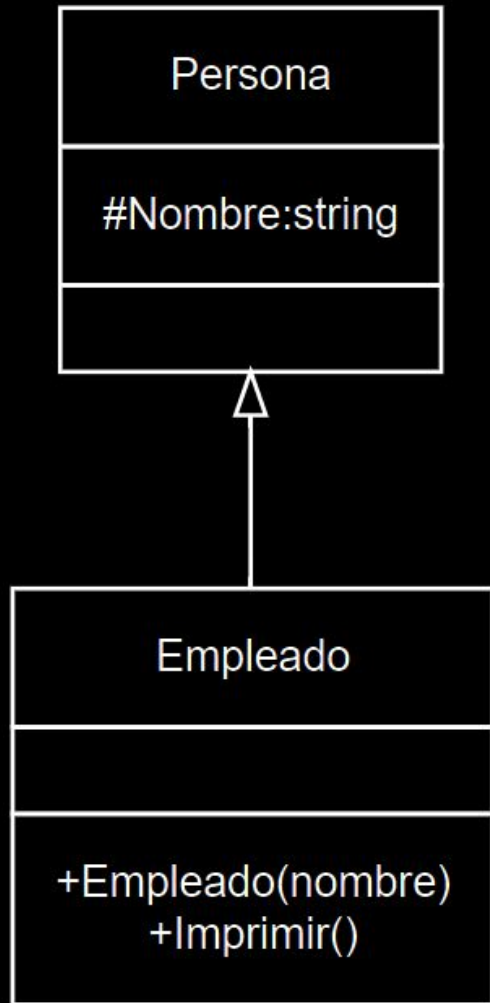
Vamos a presentar el concepto por medio de un ejemplo



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



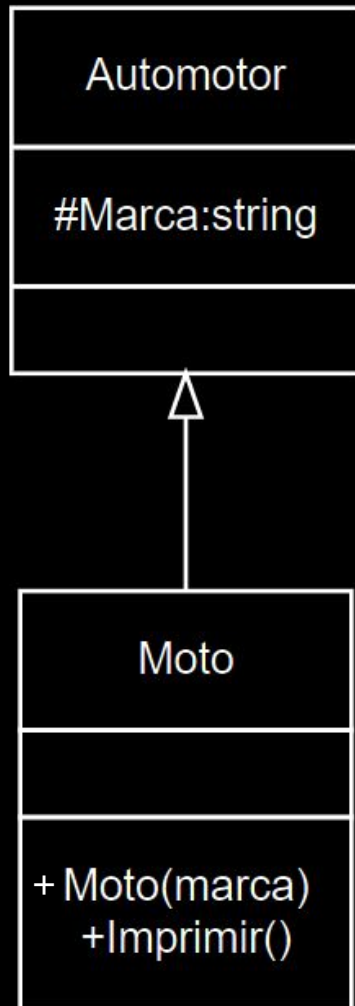
Codificar las clases Persona y Empleado



- 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



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

Interfaces - Presentación de caso

```
class Persona
{
    protected string Nombre;
}

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

class Automotor
{
    protected string Marca;
}

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



Invocar el método Imprimir de todos los elementos del vector



```
class Program
{
    static void Main(string[] args)
    {
        object[] vector = new object[3];
        vector[0] = new Moto("Zanella");
        vector[1] = new Empleado("Juan");
        vector[2] = new Moto("Gilera");
        foreach (object o in vector)
        {
            . . .
        }
    }
}
```

Completar

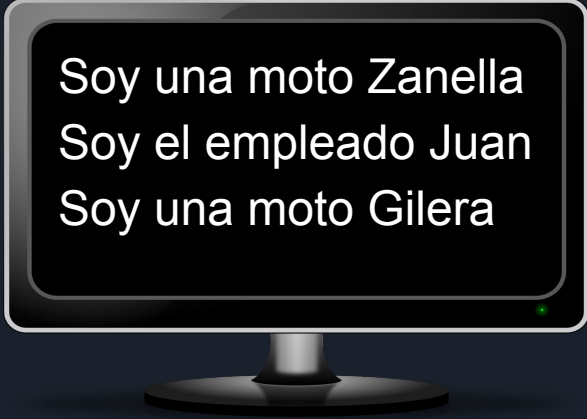


Posible solución

```
...  
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)  
{  
    (o as Empleado).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



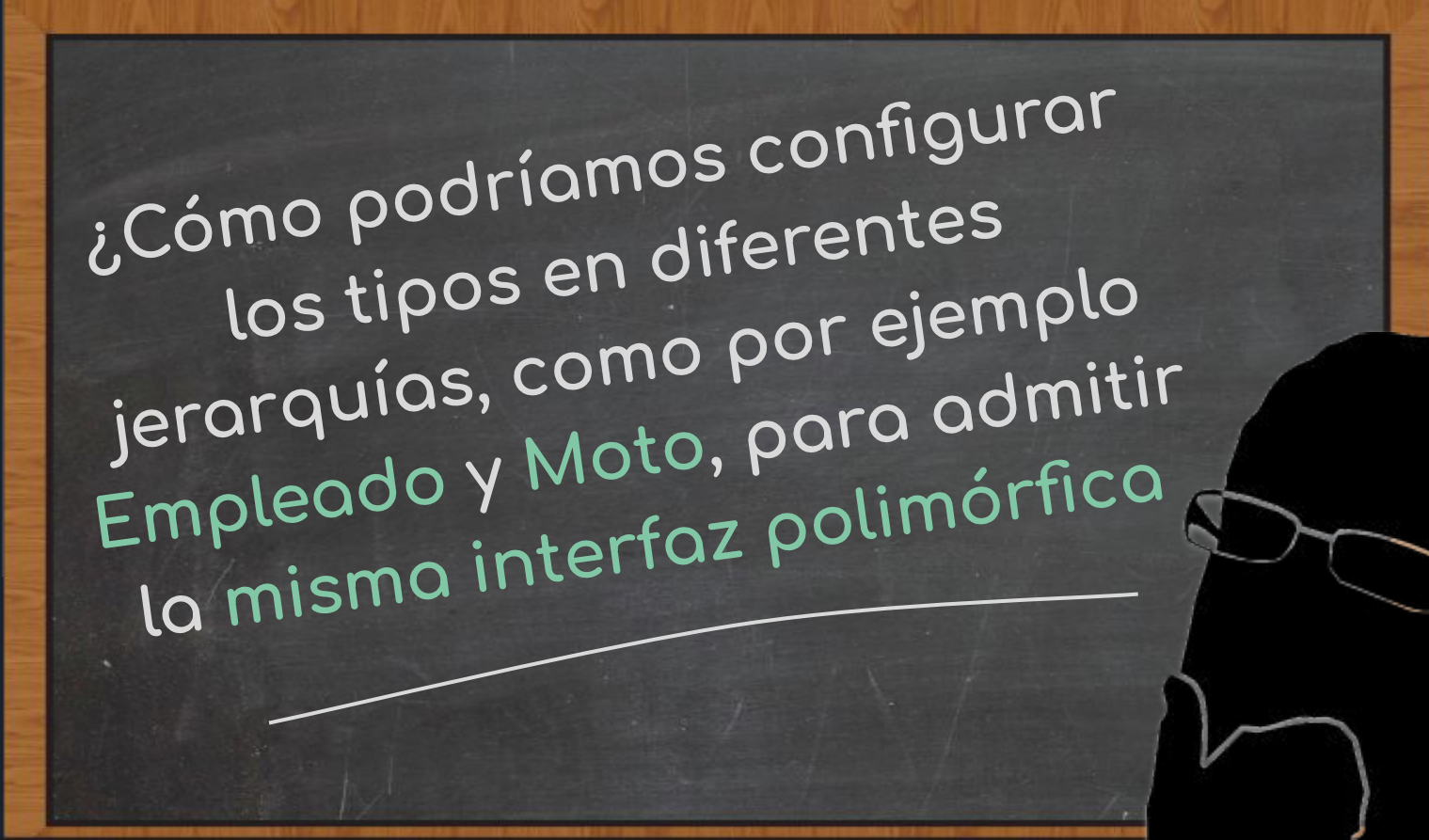
Dificultad para usar polimorfismo

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**.



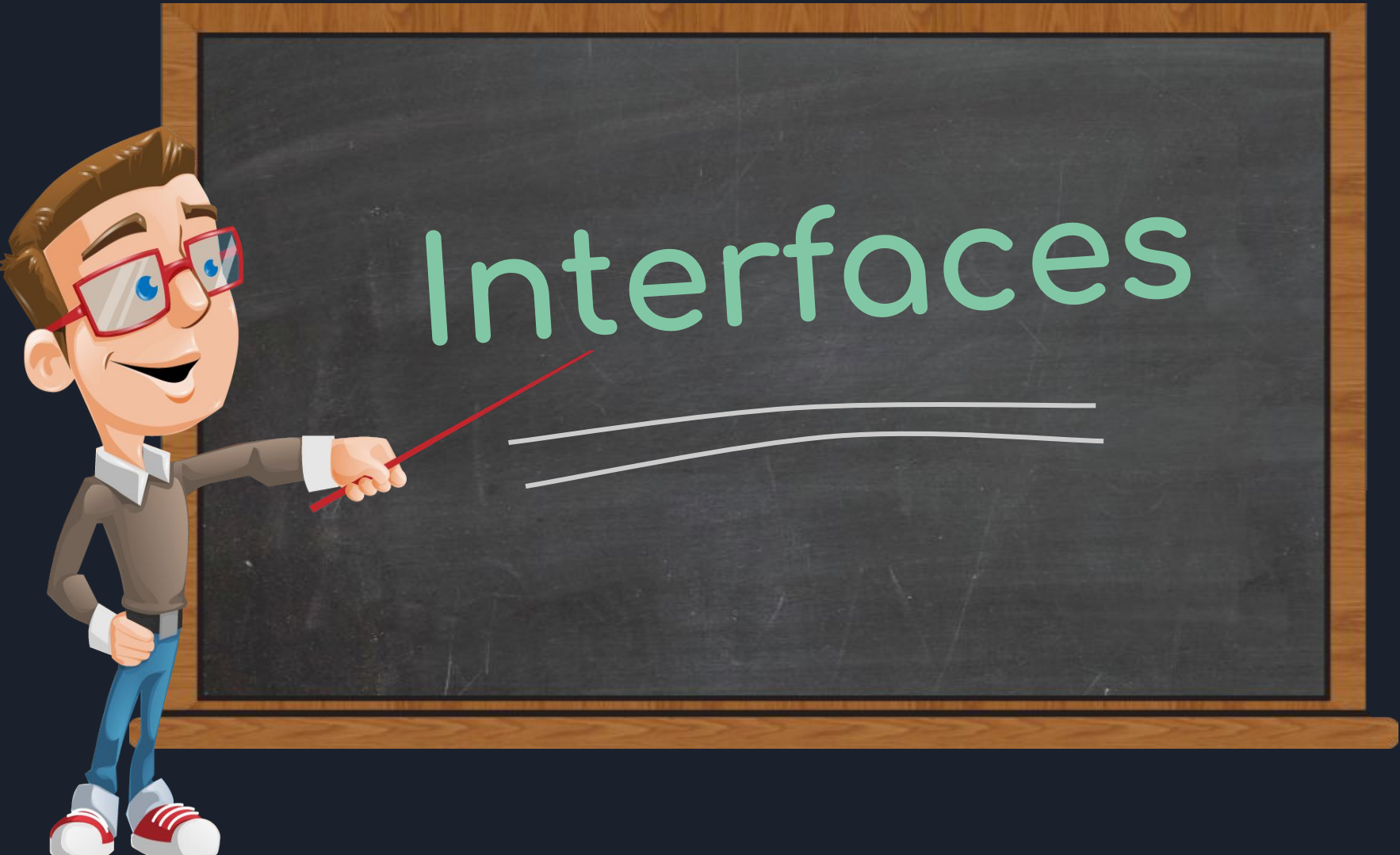
Interrogante



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



Respuesta



¿ Qué es una Interface ?

- 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 Interface ?

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

Declarando una interface

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 si quiera **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

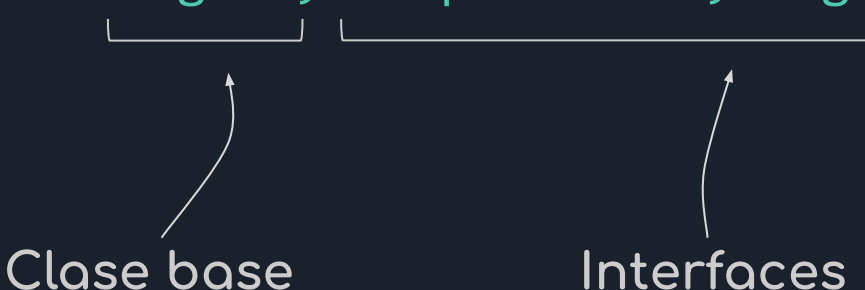
Implementación de interfaces

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

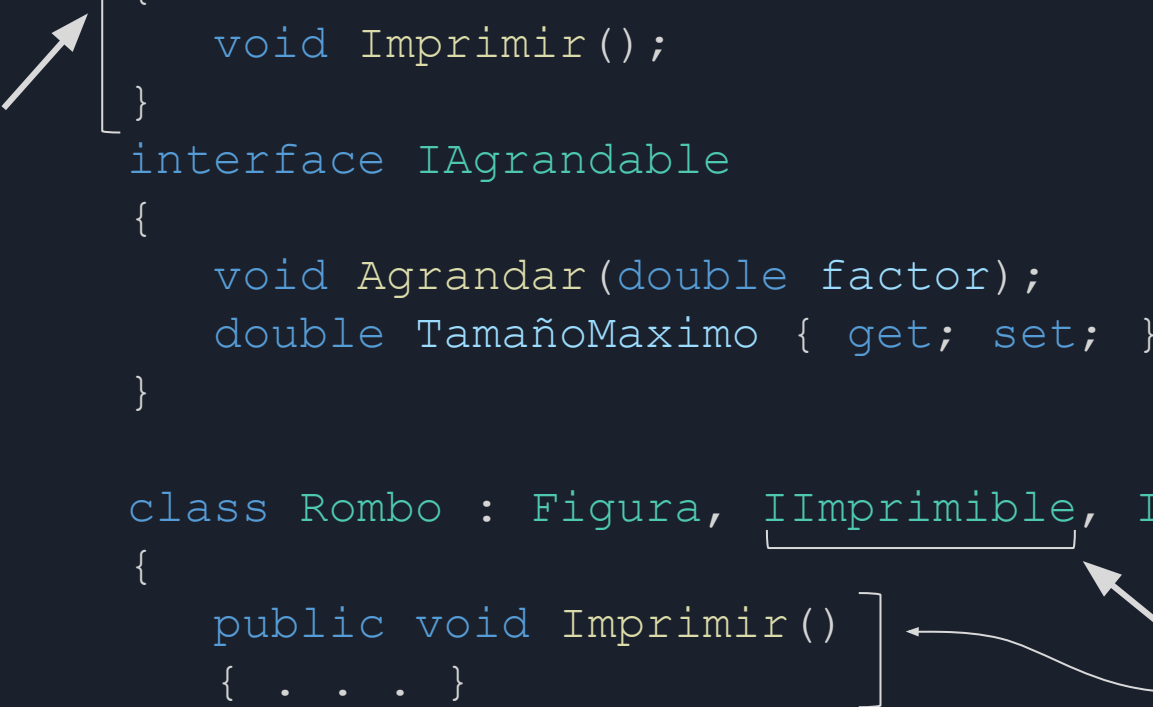
Si una clase deriva de otra clase y además implementa algunas interfaces, la clase debe ser la primera en la lista después de los dos puntos

```
class Rombo : Figura, IImplrimible, 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 IAgandable
{
    void Agrandar(double factor);
    double TamañoMaximo { get; set; }
}
```

```
class Rombo : Figura, IImprimible, IAgandable
{
    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 Interface 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



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



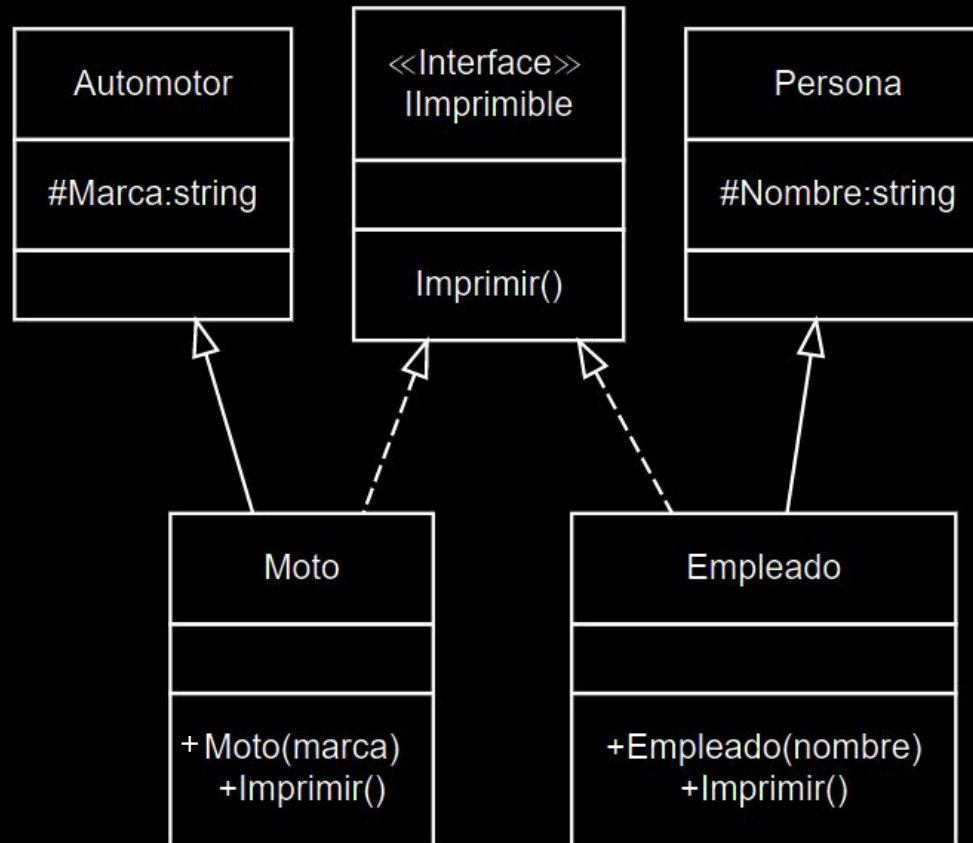
```
class Moto : Automotor, IImprimible  
{  
    . . .  
}
```



```
class Empleado : Persona, IImprimible  
{  
    . . .  
}
```



Resolución del ejercicio inicial utilizando interfaces





Solución polimórfica



```
static void Main(string[] args)
{
    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)

```
static void Main(string[] args)
{
    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");
}
```

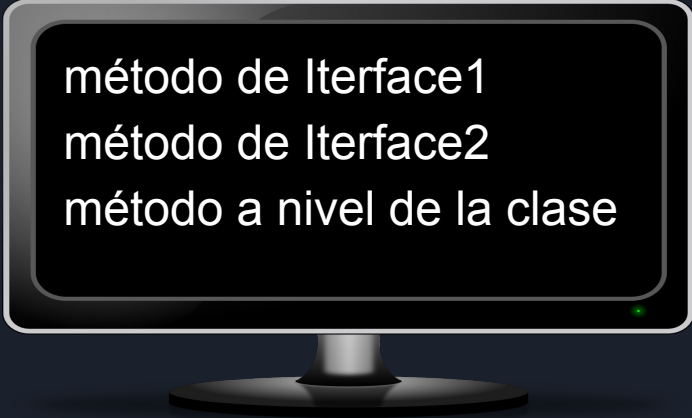


IMPORTANTE:

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 **interface**, 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

```
. . .  
static void TestOrdenamiento1()  
{  
    ArrayList lista =  
        new ArrayList() { 27, 5, 100, -1, 3 };  
    lista.Sort();  
    foreach (int i in lista)  
    {  
        Console.WriteLine(i);  
    }  
}  
. . .
```



Interface IComparable. Ejemplo de ordenamiento

El método `Sort()` de `ArrayList` funciona correctamente porque todos los elementos de la lista (en este caso de tipo `int`) son comparables entre sí porque implementan la interface `IComparable`



Ordenamiento Ejemplo 2

```
static void TestOrdenamiento2()  
{  
    ArrayList lista = new ArrayList() {  
        new Empleado("Juan"),  
        new Empleado("Adriana"),  
        new Empleado("Diego")  
    };  
    lista.Sort();  
    foreach (Empleado e in lista)  
    {  
        e.Imprimir();  
    }  
}
```


Interfaces - System.IComparable

```
24 static void testOrdenamiento2()  
25 {  
26     ArrayList lista = new ArrayList() {  
27         new Empleado("Juan"),  
28         new Empleado("Adriana"),  
29         new Empleado("Diego")  
30     };  
31     lista.Sort();
```

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.Array.SorterObjectArray.SwapIfGreaterWithItems(Int32 a, Int32 b)
en System.Array.SorterObjectArray.IntroSort(Int32 lo, Int32 hi, Int32 depthLimit)
en System.Array.SorterObjectArray.IntrospectiveSort(Int32 left, Int32 length)

```
32     foreach (Empleado e in lista)  
33     {
```

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

Interface IComparable

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

El método `Sort()` de un `ArrayList`
puede ordenar elementos de
cualquier tipo, sólo se necesita que le
enseñemos a compararse,
implementando `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

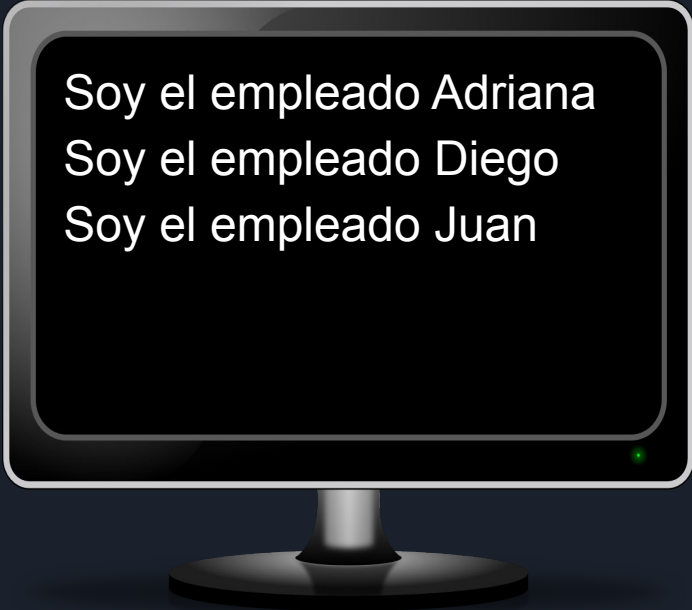
```
class Empleado : Persona, IImprimible, IComparable
{
    public int CompareTo(object obj)
    {
        string st1 = this.Nombre;
        string st2 = (obj as Empleado).Nombre;
        return st1.CompareTo(st2);
    }
    . . .
}
```

Solución: Implementar la interface
IComparable en la clase Empleado

Ordenamiento Ejemplo 2



```
static void TestOrdenamiento2()
{
    ArrayList lista = new ArrayList() {
        new Empleado("Juan"),
        new Empleado("Adriana"),
        new Empleado("Diego")
    };
    lista.Sort();
    foreach (Empleado e in lista)
    {
        e.Imprimir();
    }
}
```



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

Interface IComparer

Ejemplo de ordenamiento

1 reference
static void tes
{

ArrayList l
new Emp
new Emp
new Emp
};

2/3
^
v

lista.Sort();
foreach (Empleado e in lista)
{
e.Imprimir();
}
}

void ArrayList.Sort(IComparer comparer)

comparer: The System.Collections.IComparer implementation to use when comparing elements.

-or-

A null reference (Nothing in Visual Basic) to use the System.IComparable implementation of each element.

Sorts the elements in the entire System.Collections.ArrayList using the specified comparer.

Si queremos otro criterio de orden, podemos utilizar una sobrecarga del método `Sort` que espera como argumento un objeto comparador que debe implementar la interface `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.
        //
        int Compare(object x, object y);
    }
}
```

Ordenamiento Ejemplo 3

```
class Empleado : Persona, IImprimible, IComparable
{
    public int Legajo { get; set; }
    public void Imprimir()
    {
        Console.WriteLine($"Soy el empleado {Nombre}");
        Console.WriteLine($"Legajo: {Legajo}");
    }
    . . .
}
```

Agregar la propiedad Legajo

Modificamos el método Imprimir() de la clase Empleado

Ordenamiento Ejemplo 3

```
using System;  
using System.Collections;
```

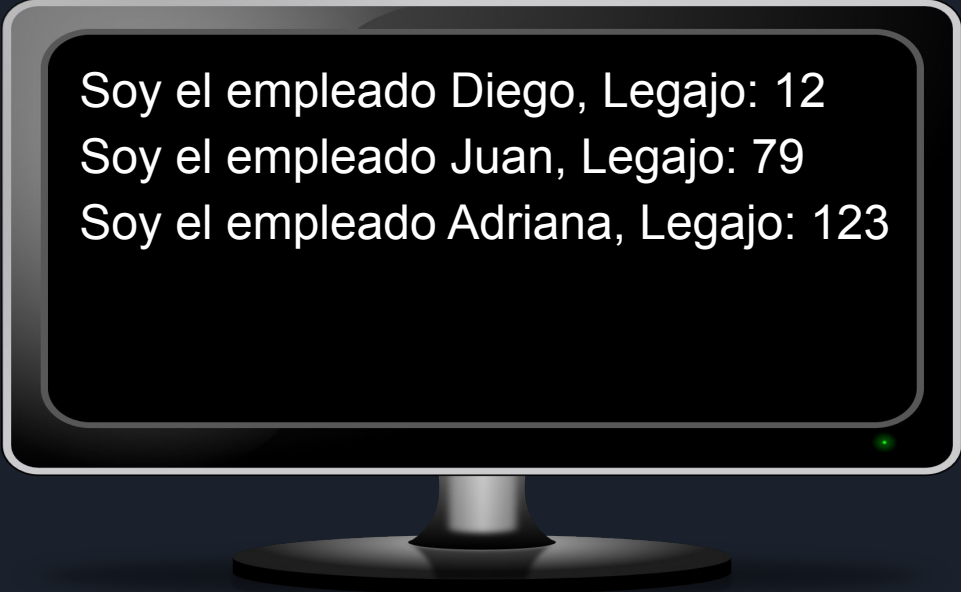
```
. . .
```

```
class ComparadorPorLegajo : IComparer  
{  
    public int Compare(object x, object y)  
    {  
        Empleado e1 = x as Empleado;  
        Empleado e2 = y as Empleado;  
        return e1.Legajo.CompareTo(e2.Legajo);  
    }  
}
```

Definir la clase
ComparadorPorLegajo e
implementar la interface
IComparer

```
static void TestOrdenamiento3()
{
    ArrayList lista = new ArrayList() {
        new Empleado("Juan") {Legajo=79},
        new Empleado("Adriana") {Legajo=123},
        new Empleado("Diego") {Legajo=12}
    };
    lista.Sort(new ComparadorPorLegajo());
    foreach (Empleado e in lista)
    {
        e.Imprimir();
    }
}
```

Ordenamiento
por legajo



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

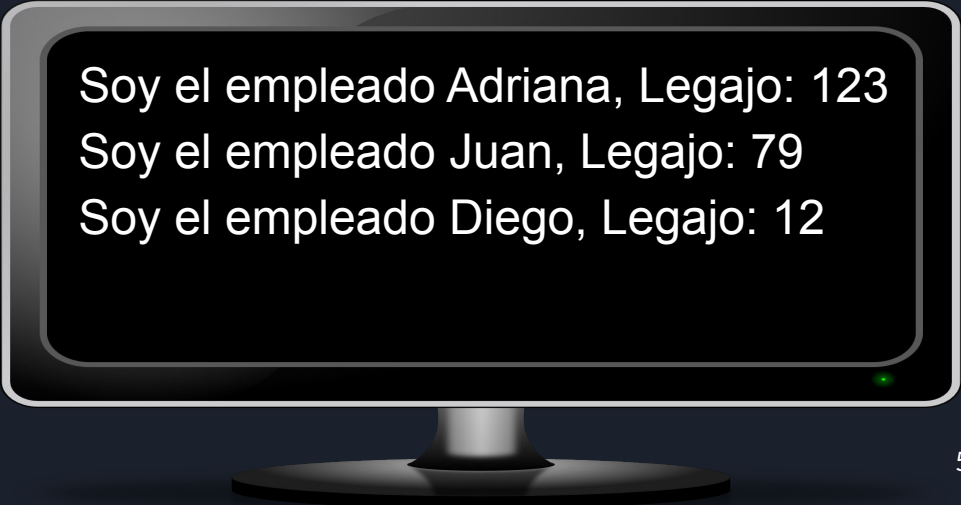
Ordenamiento - Ejemplo 4

```
class ComparadorPorLegajo : IComparer
{
    [ public bool Descendente { get; set; } = false;
    public int Compare(object x, object y)
    {
        Empleado e1 = x as Empleado;
        Empleado e2 = y as Empleado;
        int result = e1.Legajo.CompareTo(e2.Legajo);
        [ if (Descendente)
        {
            result = -result;
        }
        return result;
    }
}
```

Modificando
ComparadorPorLegajo para
permitir ordenar ascendente o
descendentemente

```
static void testOrdenamiento4()
{
    ArrayList lista = new ArrayList() {
        new Empleado("Juan") {Legajo=79},
        new Empleado("Adriana") {Legajo=123},
        new Empleado("Diego") {Legajo=12}
    };

    ComparadorPorLegajo comparador = new ComparadorPorLegajo();
    comparador.Descendente = true;
    lista.Sort(comparador);
    foreach (Empleado e in lista)
    {
        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`

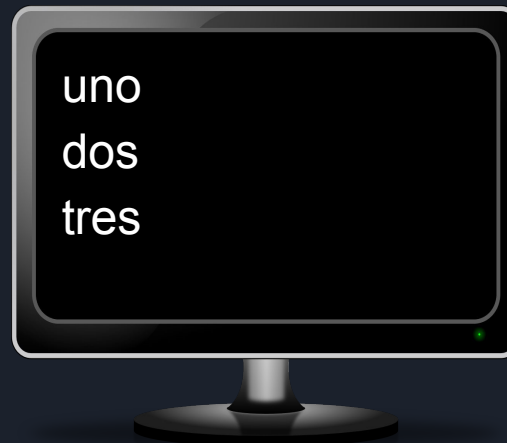
Uso de la instrucción foreach Ejemplo 1

. . .

```
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



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



Modificar el método Main y compilar



```
static void Main(string[] args)
{
    Empleado e1 = new Empleado("Juan") { Legajo = 79 };
    Empleado e2 = new Empleado("Adriana") { Legajo = 123 };
    Empleado e3 = new Empleado("Diego") { Legajo = 12 };
    Pyme miPyme = new Pyme(e1, e2, e3);
    foreach(Empleado e in miPyme)
    {
        e.Imprimir();
    }
}
```


Error de compilación

```
static void Main(string[] args)
{
    Empleado e1 = new Empleado("Juan") { Legajo = 79 };
    Empleado e2 = new Empleado("Adriana") { Legajo = 123 };
    Empleado e3 = new Empleado("Diego") { Legajo = 12 };
    Pyme miPyme = new Pyme(e1, e2, e3);
    foreach(Empleado e in miPyme)
    {
        e.Imprimir();
    }
}
```

Error de compilación:
'Pyme' no contiene ninguna definición de 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 interface `IEnumerator`



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



```
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

. . .

```
static void Main(string[] args)
```

```
{
```

```
    Empleado e1 = new Empleado("Juan") { Legajo = 79 };
```

```
    Empleado e2 = new Empleado("Adriana") { Legajo = 123 };
```

```
    Empleado e3 = new Empleado("Diego") { Legajo = 12 };
```

```
    Pyme miPyme = new Pyme(e1, e2, e3);
```

```
    foreach (Empleado e in miPyme)
```

```
    {
```

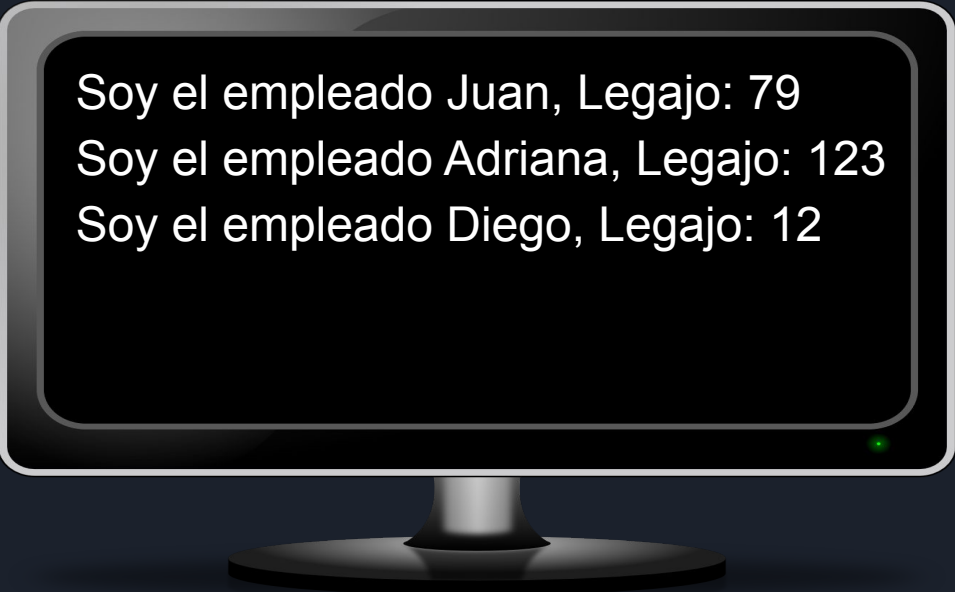
```
        e.Imprimir();
```

```
    }
```

```
}
```

. . .

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

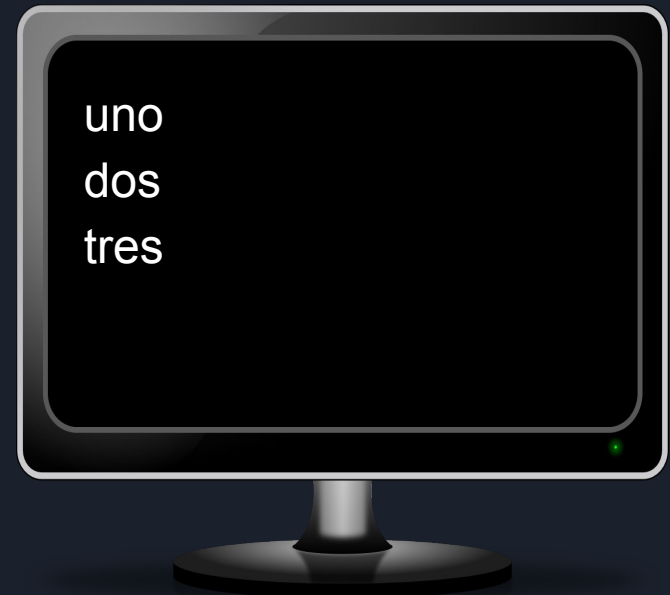
```
namespace System.Collections
{
    public interface IEnumerator
    {
        // Gets the current element in the current position.
        object Current { get; }

        // 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

```
. . .  
string[] vector = new string[]{"uno", "dos", "tres"};  
  
IEnumerator e = vector.GetEnumerator();  
while (e.MoveNext())  
{  
    Console.WriteLine(e.Current);  
}  
. . .
```



Recorriendo un enumerador

• • •

```
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

. . .

```
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 interface `System.Collections.IEnumerator` para enumerar los nombres de las estaciones del año comenzando por “verano”

Codificando un enumerador Ejemplo

```
using System;
using System.Collections;
class EnumeradorEstaciones : IEnumerator
{
    string actual = "Inicio";
    public void Reset()
    {
        actual = "Inicio";
    }
    public object Current
    {
        get
        {
            if (actual == "Inicio" || actual == "Fin")
                throw new InvalidOperationException();
            else return actual;
        }
    }
}
```

. . .


Continúa en la próxima diapositiva

Codificando un enumerador Ejemplo

. . .

```
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");
}
```

```
. . .  
static void Main(string[] args)  
{  
    IEnumerator e = new EnumeradorEstaciones();  
    while (e.MoveNext())  
    {  
        Console.WriteLine(e.Current);  
    }  
}  
. . .
```



Verano
Otoño
Invierno
Primavera

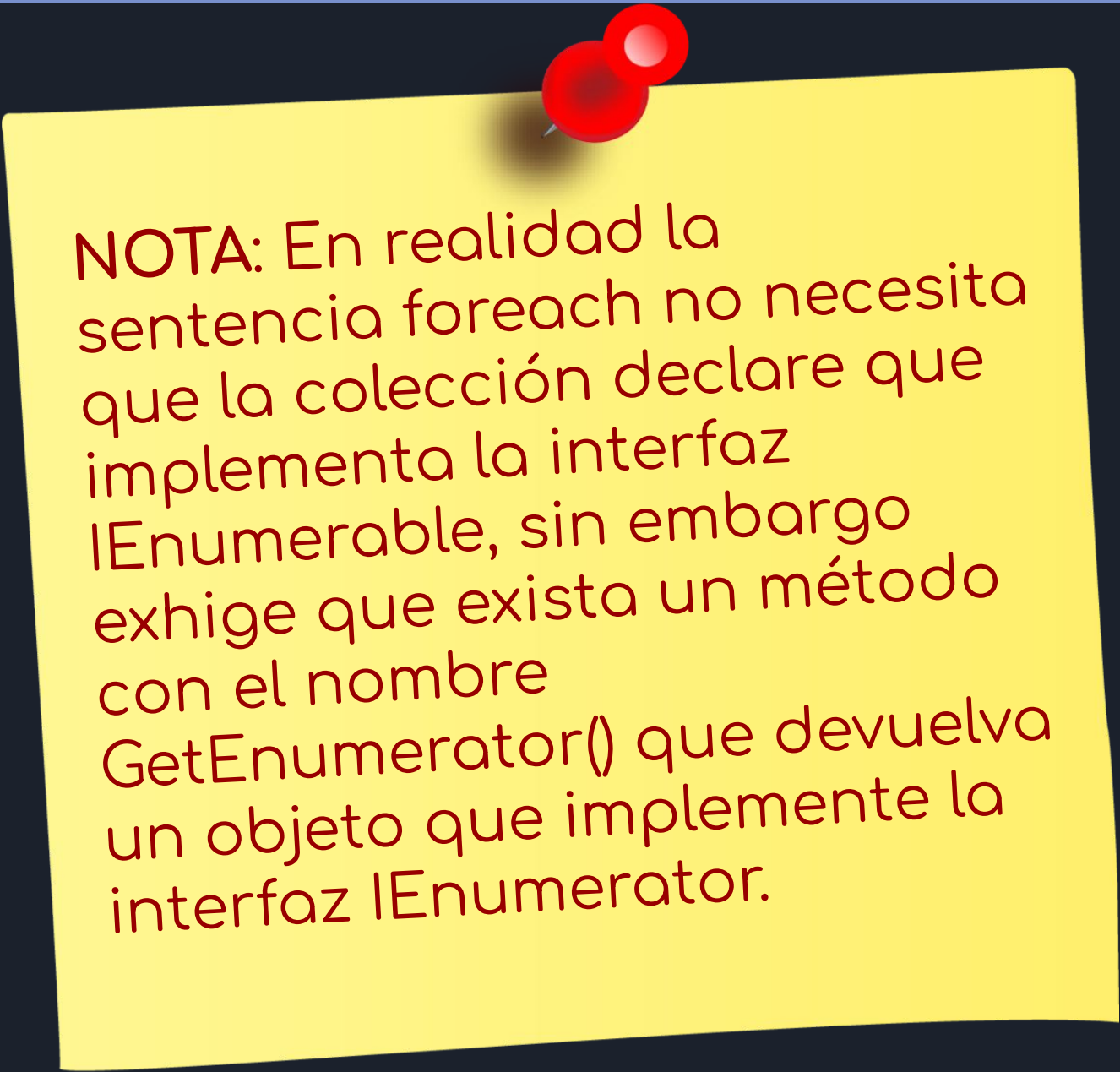
Codificando un enumerable para usar con foreach. Ejemplo

```
using System;
using System.Collections;

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

```
. . .  
static void Main(string[] args)  
{  
    Estaciones estaciones = new Estaciones();  
    foreach(string st in estaciones)  
    {  
        Console.WriteLine(st);  
    }  
}  
. . .
```





NOTA: En realidad la sentencia foreach no necesita que la colección declare que implementa 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**

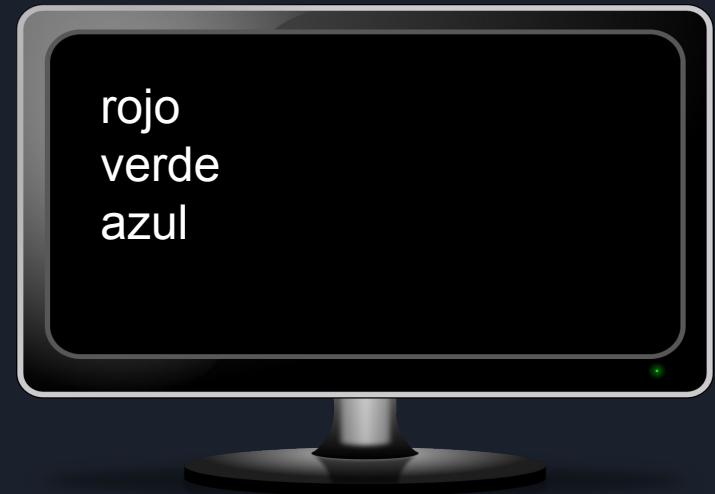
Iteradores - ejemplo 1

```
static void Main(string[] args)
{
    IEnumerator enumerador = colores();
    while (enumerador.MoveNext())
    {
        Console.WriteLine(enumerador.Current);
    }
}

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

Current es de tipo
object

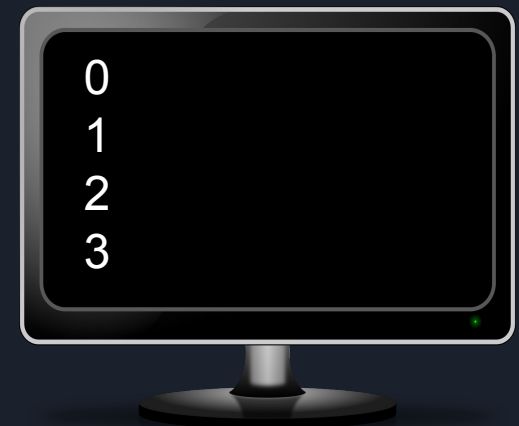
Este método es
un iterador



Iteradores - ejemplo 2

```
static void Main(string[] args) {  
    IEnumerator e = Numeros();  
    while (e.MoveNext())  
    {  
        Console.WriteLine(e.Current);  
    }  
}  
  
static IEnumerator Numeros()  
{  
    int i = 0;  
    while (true) {  
        if (i <= 3) yield return i++;  
        else yield break;  
    }  
}
```

Cuidado!
Un enumerador
generado con la
sentencia **yield** no
implementa el
método **Reset()**



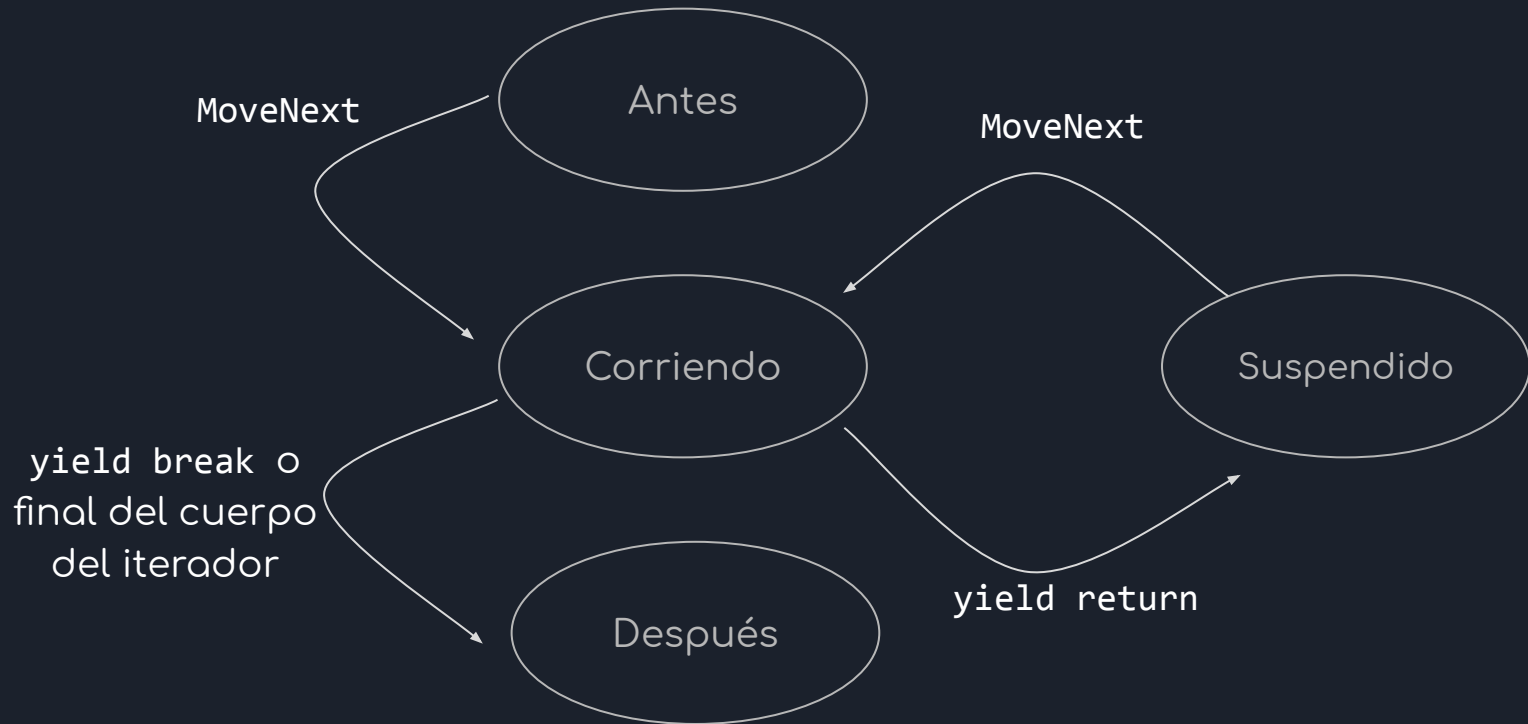
IEnumerable generado por iterador

```
static void Main(string[] args) {  
    IEnumerable poderes = poderesEstado();  
    foreach (var p in poderes)  
    {  
        Console.WriteLine(p);  
    }  
}  
  
static IEnumerable poderesEstado() {  
    yield return "Ejecutivo";  
    yield return "Legislativo";  
    yield return "Judicial";  
}
```



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

```
static void Main(string[] args)
{
    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:

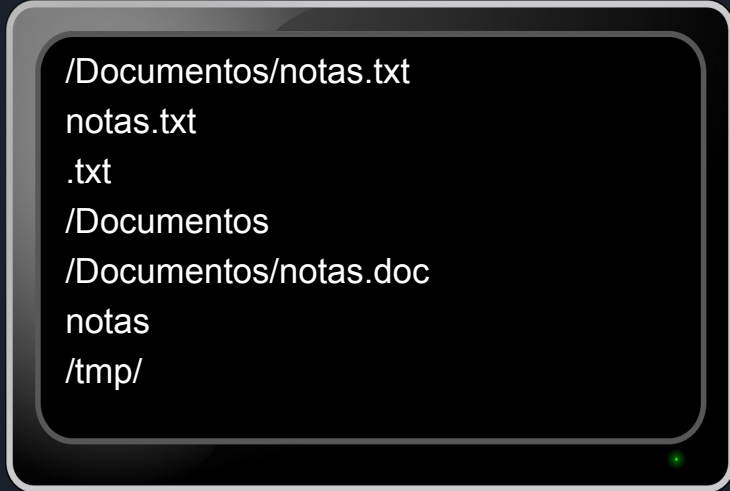
```
string archivo = "/Documentos/notas.txt";  
Console.WriteLine(Path.GetFullPath(archivo));  
Console.WriteLine(Path.GetFileName(archivo));  
Console.WriteLine(Path.GetExtension(archivo));  
Console.WriteLine(Path.GetDirectoryName(archivo));  
Console.WriteLine(Path.ChangeExtension(archivo, "doc"));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

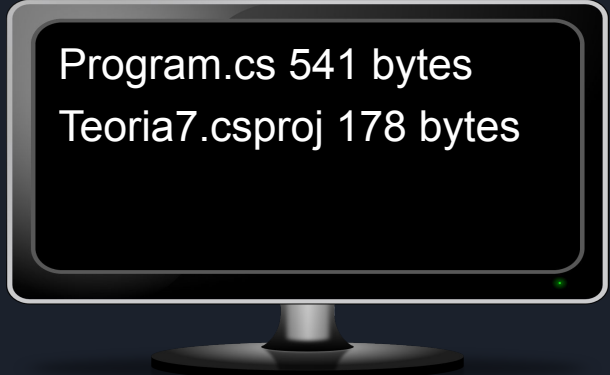


Las clases DirectoryInfo, FileInfo, Directory y File

- 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

```
static void Main(string[] args)
{
    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);
    }
}
```



Program.cs 541 bytes
Teoria7.csproj 178 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

```
public static void Main(string[] args)
{
    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()`



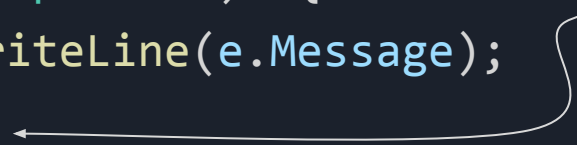
Ejemplo 2 - Manejando excepciones

```
static void Main(string[] args) {  
    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 {  
        if (sr != null) sr.Dispose();  
        if (sw != null) 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 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

```
static void Main(string[] args)
{
    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