Tema 9.3

Descargar estos apuntes

Índice

Profundizando en la Programación Orientada a Objetos

Interfaces

Básicamente un Interfaz es la definición de un conjunto de interfaces de métodos, accesores o mutadores (como **Propiedades**), indizadores, etc. Es muy parecido a definir una **clase abstracta pura**, pero **sin ningún tipo de atributo** o campo,
constructor, ni modificador de acceso (public, private, etc…). Como en las clases abstractas, las interfaces son tipos referencia, no puede crearse objetos de ellas sino sólo de tipos que deriven de ellas, y participan del polimorfismo.

Pueden implementarse en muchos lenguajes OO con idénticas características:

- Es posible la **herencia múltiple** de interfaces.
- No pueden definir atributos pero sí propiedades.
- Un interfaz **puede heredar de otro interfaz**.
- Si una clase hereda de un interfaz. Esta, deberá invalidar todo lo que hayamos definido en el mismo.

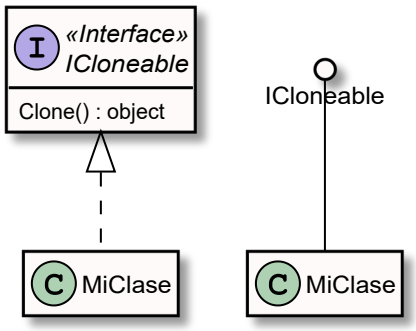
Podemos resumir diciendo que es la forma más recomendable y común de **definir la abstracción de un comportamiento**.

Interfaces en los diagramas de clases UML

Para expresar que la clase `MiClase` **implementa** el interfaz `ICloneable`.

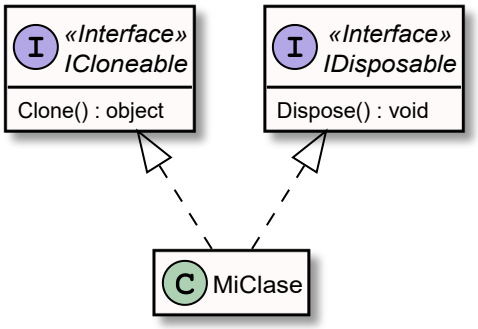
> ✋ **Nota:** Usamos la palabra **implementa** en lugar de "*hereda de*" ya que, como hemos comentado, más que responder `MiClase` a la pregunta "*es un*", un interfaz define un comportamiento abstracto que `MiClase` deberá implementar.

Podremos expresarlo de las forma siguientes formas ...



`MiClase` ahora está obligada a **implementar** el método público `Clone` con **idéntica signatura**.

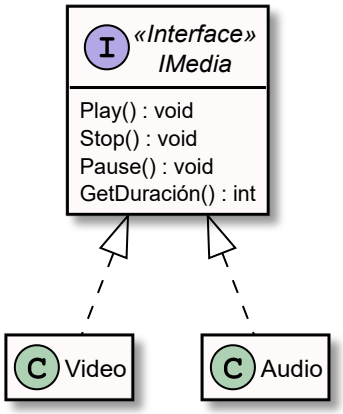
Podremos hacer que una clase **implemente** o "*herede*" de más de un interfaz.



Interfaces en C#

Como ya habrás podido apreciar en el diagrama anterior, según el convenio de nomenclatura de C#, el identificador o nombre de la clase irá siempre precedido por la letra mayúscula `I` (`I nterface`) para distinguirlo de otro tipo de clases.

```
<modificadores> interface I<identificador> : <interfacesBase>
{
    <interfaces de métodos, propiedades o indizadores>
}
```




```
interface IMedia
{
    void Play();
    void Stop();
    void Pause();
    int Duración { get; }
}
```

Para aplicar un interfaz a una clase. Haremos que esta herede del interfaz con la sintaxis de herencia que hemos usado hasta ahora.

```
class Video : IMedia
{
    // ...

    public int Duración => 0;
    public void Pause() => Console.WriteLine("Pausando el vídeo.");
    public void Play() => Console.WriteLine("Reproduciendo el vídeo.");
    public void Stop() => Console.WriteLine("Parando el vídeo.");
}
```

Interfaces de utilidad predefinidos en las BCL

Podemos decir que me permiten definir comportamientos para mi própios tipos que serán reconocidos por otras clases o tipos ya implementadas en las BCL.

> **Nota:** Podríamos utilizar interfaces própios para hacer lo mismo, **pero perderíamos interoperabilidad con el resto de clases de las BCL**.

IEnumerable

Lo veremos más adelante, al usar o definir colecciones.

ICloneable

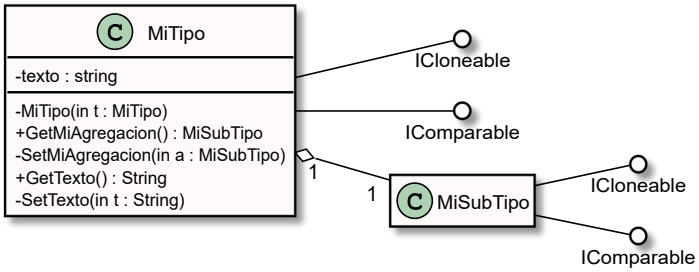
Me indicará que puedo crear copias del objeto, puesto que me obliga a implementar un "*constructor copia*" con el interfaz `object Clone()` el cual me permitirá hacer **copias en profundidad** de objetos de tipo referencia.

IComparable

Me indicará que el objeto debe implementar el método `int CompareTo(Object otro)` que me servirá para comparar dos objetos de la misma clase y que ya usamos en el **tema 7** para comparar cadenas.

> **Nota:** Recordemos brevemente que...
>
> ```
> Tipo o1 = new Tipo(...);
> Tipo o2 = new Tipo(...);
>
> // Si Tipo es IComparable entonces ...
> int comparacion = o1.CompareTo(o2);
>
> // comparacion = 0 si o1 y o2 son iguales.
> // comparacion > 0 si o1 > o2.
> // comparacion < 0 si o1 < o2.
> ```

Veamos un ejemplo "*genérico*" comentado de uso de este tipo de interfaces. Para ello, supongamos la siguiente agregación con tipos definidos por el usuario, donde ambos implementan los interfaces **`ICloneable`** e **`IComparable`**.



Vamos a suponer la clase **`MiSubTipo`** ya la hemos definido e implementa ambos interfaces. Una implementación '*genérica*' posible para **`MiTipo`** podría ser ...

```
class MiTipo : ICloneable, IComparable
{
    // Propiedad autoimplementada que representa la agregación del UML.
    public MiSubTipo MiAgregacion { get; private set; }
    // Propiedad autoimplementada que representa el atributo texto con su
    // accesor y mutador
    public string Texto { get; private set; }
    // Constructor copia privado que usaremos para hacer el clonado
    // en profundidad.
    private MiTipo(MiTipo t)
    {
        // Debo hacer un downcasting porque Clone() me devuelve un object.
        MiAgregacion = t.MiAgregacion.Clone() as MiSubTipo;
        // No hay problema en igualar los string porque es inmutable.
        Texto = t.Texto;
    }
    // Estoy forzado a implementarlo por ser ICloneable
    // Llamamos al constructor copia mediante un cuerpo de expresión.
    public object Clone() => new MiTipo(this);

    // Estoy forzado a implementarlo por ser IComparable
    public int CompareTo(object obj)
    {
        // Si obj de es de MiTipo genero una excepción.
        MiTipo t = obj as MiTipo
        ?? throw new ArgumentException("No es del tipo MiClase", "obj");
        // Para realizar la comparación voy llamando a los CompareTo de cada
        // tipo en el orden adecuado.
        int comparacion = MiAgregacion.CompareTo(t.MiAgregacion);
        if (comparacion == 0)
            comparacion = Texto.CompareTo(t.Texto);
        return comparacion;
    }
}
```

IDisposable

Me indicará que el objeto debe implementar el método `void Dispose()` que **se encargará de liberar los recursos usados por el objeto.** No confundir con el destructor `~<Tipo>()`. Indicaremos a las BCL que nuestro objeto tiene el comportamiento de liberar recursos y lo utilizaremos junto a la **instrucción** `using` la cual ya hemos usado para cerrar automáticamente los flujos al producirse una excepción cuando vimos la entrada y salida de datos. Esto es, porque en el fondo lo único que espera el `using` es que las clases marcadas con está instrucción o cláusula implementen este interfaz. En el fondo la clase base **Stream** lo implementa y por tanto tiene un método `Dispose()` que es llamado en el `finally`.

Repasando la instrucción using

- Se utiliza para instanciar objetos que contiene recursos no gestionados, esto es, que no son liberados por el recolector de basura.
- Como acabamos de comentar, estos objetos deben implementar el interfaz `IDisposable` y por tanto el método de liberación `Dispose()`.
- `using` garantiza que se llama a `Dispose()` **aunque se produzca una excepción**.

  > ✋ **Importante:** Dentro del bloque using, el objeto es de solo lectura y no se puede modificar ni reasignar puesto que dejaría de tener una referencia y no se liberaría.

Sintaxis básica:

```
// Podemos usar varios recursos liberables en el mismo ámbito así ...
using (var r1 = new TipoIDisposabe())
using (var r2 = new TipoIDisposabe())
...
using (var rN = new TipoIDisposabe())
{
    // Ámbito de uso de solo lectura de r1, r2, … , rN
}

// También podremos anidarlos.
```

En **C#8.0** evolucionó el uso de esta instrucción. De tal manera que como comentamos con los flujos, **podemos aprovechar un bloque ya definido para decidir cuando va a estar disponible un recurso**.

```
// Esté código desde C#8
if (…)
{
    using var r = new TipoIDisposabe();
    // Bloque...
}

// Equivale al siguiente en versiones anteriores...
if (…)
{
    using (var r = new TipoIDisposabe() )
    {
      // Bloque...
    }
}
```

Interpretación real de la instrucción `using`:

```
// Cuando instanciemos un objeto disposable de la siguiente manera en un método...
void Metodo()
{
    using var r = new TipoIDisposabe();
    // Cuerpo del método ...
}

// Realmente será un 'syntactic sugar' del siguiente código.
void Metodo()
{
    TipoIDisposabe r;
    try
    {
        r = new TipoIDisposabe();
        // Cuerpo del método ...
    }
    finally
    {
        if (r != null) ((IDisposable)r).Dispose();
    }
}
```

Recordemos su uso a través de un **ejemplo**...

En el ejemplo siguiente creamos una clase de utilidad para generar **logs** de nuestros programas a un determinado fichero y visualizarlos.

```
    static class Log
    {
        public static void Escribe(string fichero, string texto)
        {
5           using TextWriter w = File.AppendText($"{fichero}.log");
            w.WriteLine(DateTime.Now.ToString("dd/MM/yyy HH:mm:ss ") + texto);
        }

        public static void Muestra(string fichero)
        {
11          using TextReader r = File.OpenText($"{fichero}.log");
            string s;
            while ((s = r.ReadLine()) != null) Console.WriteLine(s);
        }
    }

    static class Ejemplo
    {
        static void Main()
        {
            Log.Escribe("DEBUG", "Empieza Main");
            Log.Escribe("DEBUG", "Finaliza Main");
            Log.Muestra("DEBUG");
        }
    }
```

Dado que las clases `TextWriter` y `TextReader` implementan la interfaz `IDisposable`, podremos usar la instrucción `using` que nos garantiza que el archivo subyacente se cierre correctamente después de las operaciones de lectura o escritura.